Tiburon: A Weighted Tree Automata Toolkit

Jonathan May and Kevin Knight

Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292
{jonmay, knight}@isi.edu

Abstract. The availability of weighted finite-state string automata toolkits made great advances in natural language processing possible. However, recent advances in model design are unsuitable for these toolkits. To combat this problem we introduce a weighted finite-state *tree* automata toolkit, which incorporates recent developments in weighted tree automata theory and is useful for natural language applications such as machine translation, sentence compression, question answering, and many more.

1 Introduction

The development of well-founded models of natural language processing applications has been greatly accelerated by the development of toolkits for finite state automata. The influential observation of Kaplan & Kay, that cascades of phonologival rewrite rules could be expressed as regular relations (equivalent to finite-state transducers) [1], was exploited by Koskenniemi in his development of the two-level morphology and accompanying system for its representation [2]. This system, which was a general program for analysis and generation of languages, pioneered the field of finite-state toolkits [3].

Successive versions of the two-level compiler such as that written by Karttunen and others at Xerox [4] were used for large-scale analysis applications in many languages [3]. At the same time, work with these systems led researchers to attack various difficulties in using them and thus contribute to the development of finite-state compilation theory. This continued work, such as that by Karttunen in intersecting composition [5] and replacement [6, 7], eventually led to the development of the Xerox finite-state toolkit, which superseded the functionality and use of the two-level tools [3].

Meanwhile, interest in adding uncertainty to finite-state models grew alongside increased availability of large datasets and increased computational power. Ad-hoc methods and individual implementations were developed for integrating uncertainty into finite-state representations [8, 9], but the need for a general-purpose weighted finite-state toolkit was clear [10]. Mohri and others at AT&T led the way with their FSM Library [11] which represented weighted finite-state automata by incorporating the theory of semirings over rational power series cleanly into the existing automata theory. Other toolkits such as van Noord's FSA utilities [12], the RWTH toolkit [13], and the USC/ISI Carmel toolkit [14] provided various interfaces and utilities for working with weighted finite-state automata. As in the unweighted case, the availability of this software led to many research projects that took advantage of pre-existing implementations [15–17] and the development of the software led to the invention of new algorithms and theory [18, 19].

While very robust and capable of development of a wide array of useful applications in NLP and beyond, these toolkits all suffer from the limitation that they can only operate on string-based regular languages. In the 1990s this was begrudgingly accepted as sufficient - the power of computers and the relatively limited availability of data prevented any serious consideration of weighted automata of greater complexity, even though more complex automata models that better captured the syntactic nature of language had long been proposed [20]. As NLP research progressed and computing power and available data increased, researchers started creating serious probabilistic tree-based models for such natural language tasks as translation [21–23], summarization [24], paraphrasing [25], language modeling [26], and others. And once again, software implementations of these models were individual, one-off efforts that took entire theses worth of work to create [27]. An extension of the AT&T toolkit that uses approximation theory to represent higher-complexity structure such as context-free grammars in the weighted finite-state string automata framework was useful for handling certain representations [28], but to truly capture tree models a tree automata framework is required.

Knight and Graehl [29] put forward the case for the top-down tree automata theory of Rounds [20] and Thatcher [30] as a logical sequel to weighted string automata for NLP. All of the previously mentioned tree-based models fit nicely into this theory. Additionally, as Knight and Graehl mention [29], most of the desired general operations in a general weighted finite-state toolkit are applicable to top-down tree automata.

We thus propose and present a toolkit designed in the spirit of its predecessors but with the tree, not the string, as its basic data structure. Tiburon is a toolkit for manipulation of weighted top-down tree automata. It is designed to be easy to construct automata and work with them – after reading this article even a linguist with no computer science background or a computer scientist with only the vaguest notions of tree automata should be able to write basic acceptors and transducers and understand how to make them more complex. To achieve these goals we have maintained simplicity in data format design, such that acceptors and transducers are very close to the way they appear in tree automata literature. We also provide a small set of generic but powerful operations that allow robust manipulation of data structures with simple commands that anyone can learn. In subsequent sections we present an introduction to the formats and operations in the Tiburon toolkit and demonstrate the powerful applications that can be easily built.

2 Related Work

The rich history of finite-state string automata toolkits was described in the previous section. Tree automata theory is extensively covered in [31]. Another treatment can be found in [32]. Timbuk [33] is a toolkit for unweighted finite state tree automata that has been used for cryptographic analysis. It is based on ELAN [34], a term rewriting computational system. MONA [35] is an unweighted tree automata tool aimed at the logic community.

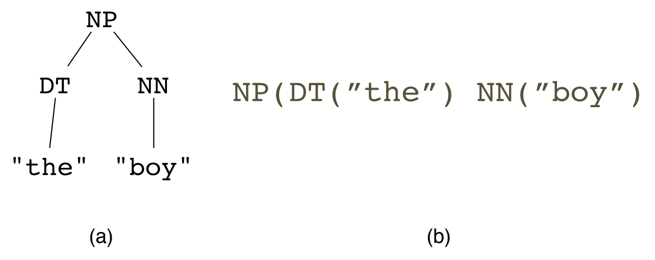


Fig. 1. (a) A typical syntax tree, and (b), its Tiburon representation

3 Trees

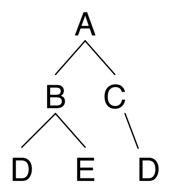
The tree is the basic data structure of tree automata. It is visually equivalent to a treelike directed labelled graph. Formally, a tree is constructed from a ranked alphabet, which we will refer to as Σ. Each member of the alphabet is assigned a non-negative integer *rank* m, and Σ_m refers to all $x \in \Sigma$ with rank m. A tree over the alphabet Σ is thus defined as:

- x, where $x \in \Sigma_0$, or
- $x(t_1, ...t_m)$, where $x \in \Sigma_m$ and $t_1, ..., t_m$ are trees over Σ.

Figure 1 shows a typical tree and its representation in Tiburon.

```
q
q -> A(q r) # 0.8
q -> B(r E) # 0.2
r -> C(r)
r -> D
```

(a)



(b)

Fig. 2. (a) A regular tree grammar, and (b), a tree in the grammar's language

4 Regular Tree Grammars

As finite-state string acceptors have an equivalent grammar, the regular string grammar, so do finite-state tree acceptors have the equivalent regular tree grammar. Tree automata literature favors the grammar representation, as tree acceptors must be written as hypergraphs, and this can be confusing. Regular tree grammars look very similar to context-free grammars (in fact, a context-free grammar is a special case of a regular tree grammar) and thus tend to be a very comfortable formalism. Analogous to their string counterpart, weighted regular tree grammars recognize a weighted, possibly infinite language of trees. Formally, a weighted regular tree grammar over Σ and under semiring $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$ consists of a finite set N of *nonterminal symbols* disjoint from Σ, a start symbol $s \in N$, and a set P of *productions* of the form $a \rightarrow r, \delta$, where $a \in N$, r is a tree over $\Sigma \cup N$, and $\delta \in \mathbb{K}$ is the associated weight. Informally, a regular tree grammar "works" similar to a context-free grammar in that from the start symbol, a sequence of rewrites is performed, replacing nonterminals with trees as specified by the productions, until the generated/recognized tree has no nonterminals remaining. The weight of the tree, w, is determined by starting with $\bar{1}$ and for each production $p = a \rightarrow r, \delta$ used, setting $w = w \otimes \delta$. Figure 2(a) shows a typical weighted grammar in Tiburon format. When weights are omitted on productions, a weight of $\bar{1}$ is assumed. Figure 2(b) shows a tree that this grammar recognizes. If we use the probability semiring, the tree has an associated weight of 0.16

```
qe
qe -> A(qe qo) # .1
qe -> A(qo qe) # .8
qe -> B(qo) # .1
qo -> A(qo qo) # .6
qo -> A(qe qe) # .2
qo -> B(qe) # .1
qo -> C # .1
```

(a)

```
B(C): 0.0100
A(C B(C)): 0.0008
B(A(C C)): 0.0006
A(B(C) C): 0.0001
B(B(B(C))): 0.0001
```

(b)

Fig. 3. (a) `even.rtg`, and (b) its top 5 dervations

4.1 Generation

A fundamental operation on a regular tree grammar is the generation of trees that are in the grammar's language. In many cases it is desirable to obtain a list of trees, ordered by weight. Syntax machine translation applications often produce translations by first obtaining a list of the top 25,000 trees of English translations from their translation models, and then using that list in a subsequent reranking operation. The `-k` operation invokes the k-best paths algorithm for a hypergraph, as described by Huang and Chiang [36]. For example, given the grammar `even.rtg` depicted in figure 3(a), we issue this command:

```
java -jar tiburon.jar -k 5 even.rtg
```

The five derivations with highest probability in the grammar are returned, as depicted in figure 3(b).

```
q
q -> S(np vp)
np -> NP(dt nn)
np -> NP(dt jj nn)
dt -> the
dt -> a
jj -> funny
jj -> blue
jj -> strange
nn -> fish
nn -> carrot
vp -> VP(v np)
v -> ate
v -> created
```

(a)

```
S(NP(the carrot) VP(ate NP(a fish))): 1.0000
S(NP(the fish) VP(created NP(a carrot))): 1.0000
S(NP(a carrot) VP(created NP(the funny fish))): 1.0000
S(NP(the fish) VP(created NP(a carrot))): 1.0000
S(NP(a fish) VP(created NP(a fish))): 1.0000
```

(b)

Fig. 4. (a) `vic.rtg`, and (b) five random dervations

Another generation operation supported, the `-g` operation, stochastically returns trees from a grammar, probabilistically choosing states to expand until a tree is obtained or a threshold of expansion is reached. This operation is useful for experimenting with the effects of uncertainty on other automata. For example, one could use the stochastic generation from a language model as an input to a transliterating transducer to judge the latter's ability to transliterate when presented with unexpected input. Given the grammar `vic.rtg`, depicted in figure 4(a), we issue this command:

```
java -jar tiburon.jar -g 5 vic.rtg
```

Five random derivations are returned, as seen in figure 4(b).

4.2 Intersection

A cricital advantage of the automata abstraction of building application models is the ability to subdivide large problems into smaller ones. To that end, a useful manipulation of regular tree grammars is their intersection – several grammars that each define a small portion of a language can then be intersected to form a grammar that represents the entire language. It is advantageous that we are working with regular tree grammars, which, as noted in [29], are closed under intersection. As an example, consider the grammar, `even.rtg`, described above, which produces trees with an even number of labels. The grammar `three.rtg`, depicted in figure 5(a), produces trees with a number of

```
q3
q3 -> A(q1 q1) # .25
q3 -> A(q3 q2) # .25
q3 -> A(q2 q3) # .25
q3 -> B(q2)    # .25
q2 -> A(q2 q2) # .25
q2 -> A(q1 q3) # .25
q2 -> A(q3 q1) # .25
q2 -> B(q1)    # .25
q1 -> A(q3 q3) # .025
q1 -> A(q1 q2) # .025
q1 -> A(q2 q1) # .025
q1 -> B(q3)    # .025
q1 -> C        # .9
```

(a)

```
qe_q3
qe_q3 -> A(qo_q1 qe_q1) # 0.2000
qe_q3 -> A(qo_q3 qe_q2) # 0.2000
qe_q3 -> A(qo_q2 qe_q3) # 0.2000
qe_q3 -> B(qo_q2) # 0.0250
qe_q3 -> A(qe_q1 qo_q1) # 0.0250
qe_q3 -> A(qe_q3 qo_q2) # 0.0250
qe_q3 -> A(qe_q2 qo_q3) # 0.0250
qe_q2 -> A(qe_q3 qo_q1) # 0.0250
qe_q2 -> A(qe_q1 qo_q3) # 0.0250
qe_q2 -> A(qe_q2 qo_q2) # 0.0250
qe_q2 -> A(qo_q3 qe_q1) # 0.2000
qe_q2 -> A(qo_q1 qe_q3) # 0.2000
qe_q2 -> A(qo_q2 qe_q2) # 0.2000
qe_q2 -> B(qo_q1) # 0.0250
```

(b)

Fig. 5. (a) `three.rtg`, and (b) a portion of the intersection of `two.rtg` and `three.rtg`. The complete grammar has 43 rules.

labels divisible by three. We obtain a grammar which produces trees with a number of labels divisible by six. by using the following command. A portion of that grammar is shown in figure 5(b).

```
java -jar tiburon.jar even.rtg three.rtg
```

A more practical use of intersection might be the intersection of a grammar representing an English syntax language model with a grammar representing candidate English tree translations of a Chinese string. The intersection produces the candidate translations, reweighted by the language model.

```
t
t -> D(q r) # 0.2
t -> D(q s) # 0.3
q -> A # 0.3
r -> B # 0.2
s -> B # 0.6
s -> C # 0.4
```

(a)

```
D(A B): 0.0540
D(A C): 0.0360
D(A B): 0.0120
```

(b)

```
Warning: returning fewer
trees than requested
D(A B): 0.0660
D(A C): 0.0360
```

(c)

Fig. 6. (a) Undeterminized grammar `undet.rtg`, (b) *k*-best list without determinization, and (c) *k*-best list with determinization

4.3 Weighted Determinization

For a variety of reasons, regular tree grammars often can produce the same tree with more than one derivation and, consequently, more than one weight. This property is undesirable to users wishing to know, for example, the total probability of a particular tree in a language. It is also undesirable to have repeated results in a k-best list. The `-d` operation invokes May and Knight's weighted determinization algorithm for tree automata [37]. As an example, consider the grammar, `undet.rtg`, depicted in figure 6(a). This grammar has two differently-weighted derivations of the tree `D(A B)`, as can be seen when the top three derivations are obtained, as in figure 6(b). The following command determinizes the grammar and attempts to return the top three derivations in the determinized grammar:

```
java -jar tiburon.jar -d 5 -k 3 undet.rtg
```

Of course, since there are now only two derivations, only two trees are returned, as seen in figure 6(c). The `-d 5` argument signifies the maximum time allowed for determinization, in minutes. Determinization is, in the worst case, an exponential-time operation, so it is helpful in practical matters to have the ability to abort lengthy operations.

4.4 Pruning

In real systems using large grammars to represent complex languages, resource management is a very real issue. Even as computers increase in power, the added complexity of tree automata forces practitioners to combat exponential-time processes. One way of avoiding long running times is to avoid working with larger automata than are strictly necessary. A common technique for pruning finite-state (string) automata is to use the forward-backward algorithm to calculate the highest-scoring path each arc in the automaton is involved in, and then prune the arcs that are only in low-scoring paths [38]. In this manner, automata may be shrunk without affecting the best, and likely most useful, recognized paths.

We adapt this technique for tree automata by using the inside-outside algorithm, which is the appropriate dual to the forward-backward algorithm for tree structures [39, 40]. The `-p` option removes productions from a tree grammar that are involved in paths worse than some threshhold. We demonstrate pruning using the `-c` option to look at an overview of a grammar. The file `c1s4.determ.rtg` represents a language of possible translations of a chinese sentence. We inspect the grammar as follows:

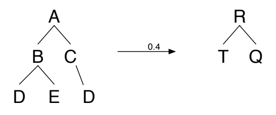
```
java -jar tiburon.jar -m tropical -c c1s4.determ.rtg

Check info:

        113 states

        168 rules

        28 unique terminal symbols

        2340 derivations
```

The `-m tropical` flag is used because this grammar is weighted in the tropical semiring. We prune the grammar and then inspect it as follows:

```
java -jar tiburon.jar -m tropical -p 8 -c c1s4.determ.rtg

Check info:

        111 states

        158 rules

        28 unique terminal symbols

        780 derivations
```

The option `-p x` where x is some number and γ_0 is the weight of the best-scoring derivation in the grammar, means that all rules with an inside-outside cost above the threshold $\kappa = \gamma_0 \otimes x$ will be pruned. Since we are in the tropical semiring and the weights are represented as a natural log, this means all productions that are in derivations $e^8 \approx 2980$ times less likely than the best derivation or more are pruned. A quick check of the top derivations (using `-k` and *diff*) shows that the grammars do not differ in their derivation lists until the 455th-highest derivation.

```
q
q.A(x0: x1:) -> R(r.x1 r.x0) # 0.4
q.A(x0: x1:) -> R(r.x0 s.x1) # 0.6
q.C(x0:) -> L(q.x0)
r.C(x0:) -> T
r.B(x0: x1:E) -> Q
s.B -> X
```
(a)



(b)

Fig. 7. (a) A tree-tree transducer, and (b), a possible weighted transduction

5 Tree Transducers

Top-down tree transducers in Tiburon come in two varieties; *tree-to-tree* and *tree-to-string*. They represent the weighted transformation of a language of trees into either a language of trees or a language of strings, respectively. One can also think of transducers as representing a language of weighted (tree, tree) or (tree, string) pairs; this approach is natural when considering a transducer as a weighted synchronous tree substitution grammar [41, 42]. We omit a formal definition of top-down tree transducers here, as an informal approach better suits the needs of the reader wishing to become a user of the Tiburon system, however we refer the reader to [31] for a thorough treatment.

Figure 7(a) is a sample tree-to-tree transducer in Tiburon format. Like a tree grammar, it has a start symbol and a set of optionally weighted productions. A transduction operation walks down an input tree, transforming it and recursively processing its branches. For example, the first rule in figure 7(a) means: “When in state q, facing an input subtree with root symbol A and two children about which we know nothing, replace it with an output subtree rooted at R with two children. To compute the output subtree's left child, recursively process the input subtree's right child with state r. To compute the output subtree's right child, recursively process the input subtree's left child with state r.” Figure 7(b) shows a possible transduction represented by this transducer. The format for tree-to-string transducers in Tiburon is similar to that for tree-to-tree transducers; the sole differences are the right side of productions are strings, not trees, and the special symbol `*e*` representing the empty string may be used. Tiburon supports operations on both types of transducers where appropriate (e.g. composition of two tree-string transducers is, understandably, not supported).

5.1 Application

The critical function of a transducer is its ability to transduce. We call this action application, since we apply a tree to the transducer as input and obtain a regular tree grammar or context-free grammar as the output. We use the `-l` operation (for "left-side" transduction; right-side is considered a separate operation) to accomplish this. If the transducer in figure 7(a) is in a file `xr1.trans` the transduction performed in figure 7(b) can be accomplished as follows:

```
echo "A(B(D E) C(D))" | java -jar tiburon.jar -l -s xr1.trans
q2
q2 -> R(q0 q1) # 0.4000
q1 -> Q # 1.0000
q0 -> T # 1.0000
```

The `-s` flag tells Tiburon to expect the input tree from *stdin* instead of a file. It should be evident that the grammar produced generates the tree on the right side of figure 7(b).

5.2 Composition

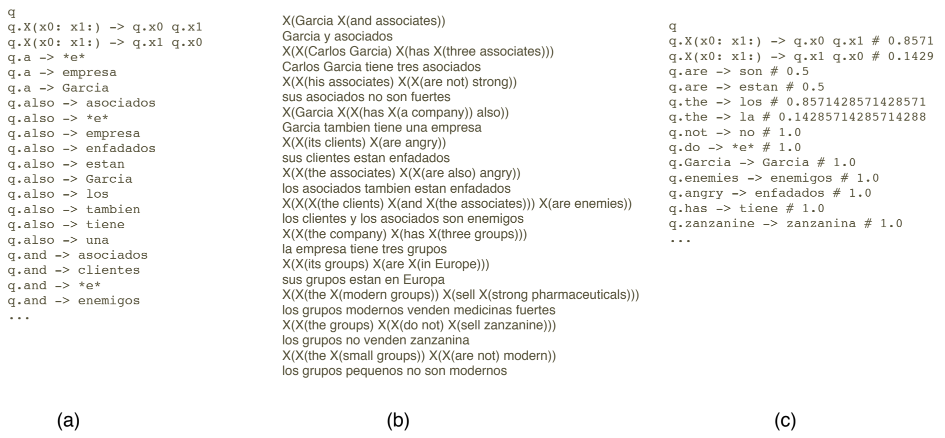
As with regular tree grammars, the ability to consider a complex transducer as several smaller transducers and then programmatically combine them is a boon to model development. Unlike string transducers, tree transducers are not always composable; that is, the transduction possible by a sequence of two tree transducers may not be possible by a single tree transducer of the same type. However, certain classes of top-down tree transducer have been shown to be composable (for example, those with no rules that cause a copying or deletion of children [29]), and thus we provide the ability to compose transducers. For example, consider the transducer below, which is in a file `xr2.trans`:

```
q
q.R(x0: x1:) -> R(R q.x1 R q.x0)
q.T -> B
```

```
q.Q -> C

q.X -> A
```

The following command composes `xr1.trans` with `xr2.trans` and passes a tree through them, returning the top (in this case the only) output tree:

```
echo "A(B(D E) C(D))" | java -jar tiburon.jar -ls -k 1 \
    xr1.trans xr2.trans
R(R C R B): 0.4000
```


(a)

```
q
q.X(x0: x1:) -> q.x0 q.x1
q.X(x0: x1:) -> q.x1 q.x0
q.a -> *e*
q.a -> empresa
q.a -> Garcia
q.also -> asociados
q.also -> *e*
q.also -> empresa
q.also -> enfadados
q.also -> estan
q.also -> Garcia
q.also -> los
q.also -> tambien
q.also -> tiene
q.also -> una
q.and -> asociados
q.and -> clientes
q.and -> *e*
q.and -> enemigos
...
```

(b)

```
X(Garcia X(and associates))
Garcia y asociados
X(X(Carlos Garcia) X(has X(three associates)))
Carlos Garcia tiene tres asociados
X(X(his associates) X(X(are not) strong))
sus asociados no son fuertes
X(Garcia X(X(has X(a company)) also))
Garcia tambien tiene una empresa
X(X(its clients) X(are angry))
sus clientes estan enfadados
X(X(the associates) X(X(are also) angry))
los asociados tambien estan enfadados
X(X(X(the clients) X(and X(the associates))) X(are enemies))
los clientes y los asociados son enemigos
X(X(the company) X(has X(three groups)))
la empresa tiene tres grupos
X(X(its groups) X(are X(in Europe)))
sus grupos estan en Europa
X(X(the X(modern groups)) X(sell X(strong pharmaceuticals)))
los grupos modernos venden medicinas fuertes
X(X(the groups) X(X(do not) X(sell zanzanine)))
los grupos no venden zanzanina
X(X(the X(small groups)) X(X(are not) modern))
los grupos pequenos no son modernos
```

(c)

```
q
q.X(x0: x1:) -> q.x0 q.x1 # 0.8571
q.X(x0: x1:) -> q.x1 q.x0 # 0.1429
q.are -> son # 0.5
q.are -> estan # 0.5
q.the -> los # 0.8571428571428571
q.the -> la # 0.14285714285714288
q.not -> no # 1.0
q.do -> *e* # 1.0
q.Garcia -> Garcia # 1.0
q.enemies -> enemigos # 1.0
q.angry -> enfadados # 1.0
q.has -> tiene # 1.0
q.zanzanine -> zanzanina # 1.0
...
```

Fig. 8. (a) Portion of a 261-rule unweighted tree-to-string transducer. (b) 15 training pairs. (c) Portion of the 31-rule weighted tree-to-string transducer produced after 20 iterations of EM training (the remaining rules had probability 0).

5.3 Training

A very common task as part of any research effort with weighted finite-state transducers is the assignment of appropriate weights to the transducer productions. Choosing these weights manually becomes infeasible when working with industrial-size transducers on naturally ocurring data. As shown by Dempster, Laird, and Rubin, the EM algorithm can be used to train a transducer by providing a corpus of data and recursively maximizing the likelihood of the corpus [43]. Carmel provides a simple interface to this algorithm for string transducers [14]. Tiburon provides the `-t` operation, which implements the techinque decribed by Graehl and Knight for training tree transducers using EM [44].

Given the 261-rule unweighted tree-to-string transducer depicted in figure 8(a) in file `y1.ts` and the 15 (tree, string) pairs in figure 8(b) in file `y1.train`, we run the command:

```
java -jar tiburon.jar -t 20 y1.train y1.trans
```

This produces the weighted tree-to-string transducer in figure 8(c).

6 Application

We will next breifly describe two natural language applications that we have built using Tiburon. The first was originally a laborious construct but has been easily and quickly replicated using Tiburon formalisms. The second is a string automata-based application, but is easily representable in Tiburon, demonstrating that Tiburon supersedes weighted finite-state string toolkits.

6.1 The Yamada Translation Model

The translation model of Yamada and Knight [22] is a specialized model for predicting a Japanese string given an English tree. The custom implementation of this model, built by Yamada as part of his PhD thesis [27], took more than one year to complete. Graehl and Knight [44] showed how this model could be represented as a four-state tree-to-string transducer. We built this transducer using simple scripting tools and trained it on the training data used by Yamada and Knight to produce alignments [22]. The complete process took only 2 days to complete. For complete results we refer the reader to [45].

6.2 Japanese Transliteration

Knight and Graehl describe a method for using a cascade of finite-state string transducers to perform English-Japanese transliteration [46]. Of course, a weighted string transducer toolkit such as Carmel would be well suited for this task, but Tiburon is suited for the job as well. By converting string transducers into monadic tree transducers, equivalent results may be obtained. We have used simple scripts to transform the string transliteration transducers into these monadic trees, and have been able to reproduce the transliteration operations. Thus, we see how Tiburon may be used for string-based applications as well as tree-based applications.

7 Future Work

We are currently working on the next version of Tiburon, which will place an emphasis on incorporating tree automata versions of the optimization techniques described in [10], particularly those for lazy composition of transducers. We will also support right-to-left application through transducers and more comprehensive error support.

8 Conclusion

We have described Tiburon, a general weighted finite-state tree automata toolkit, and described some of its functions and their applications to natural language applications. The Tiburon binaries and accompanying documentation may be downloaded from the following URL: (NOTE: need to have this?!).

References

1. Kaplan, R.M., Kay, M.: Phonological rules and finite-state transducers. In: Linguistic Society of America Meeting Handbook, Fifty-Sixth Annual Meeting, New York (1981) Abstract.
2. Koskenniemi, K.: Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki (1983)
3. Karttunen, L., Beesley, K.R.: A short history of two-level morphology. Presented at the ESSLLI-2001 Special Event titled ”Twenty Years of Finite-State Morphology” (2001) Helsinki, Finland.
4. Karttunen, L., Beesley, K.R.: Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA (1992)
5. Karttunen, L., Kaplan, R.M., Zaenen, A.: Two-level morphology with composition. In: COLING ’92, Nantes, France (1992) 141–148
6. Karttunen, L.: The replace operator. In: ACL-95 Proceedings, Boston (1995) 16–23
7. Karttunen, L.: Directed replacement. In: ACL-96 Proceedings, Santa Cruz, CA (1996)
8. Riccardi, G., Pieraccini, R., Bocchieri, E.: Stochastic automata for language modeling. Computer Speech & Language **10**(4) (1996) 265–293
9. Ljolje, A., Riley, M.D.: Optimal speech recognition using phone recognition and lexical access. In: ICSLP-1992. (1992) 313–316
10. Mohri, M., Pereira, F.C.N., Riley, M.: The design principles of a weighted finite-state transducer library. Theoretical Computer Science **231** (2000) 17–32
11. Mohri, M., Pereira, F.C.N., Riley, M.: A rational design for a weighted finite-state transducer library. In: Proceedings of the 7th Annual AT&T Software Symposium, AT&T Labs (1997)
12. van Noord, G., Gerdemann, D.: An extendible regular expression compiler for finite-state approaches in natural language processing. In Boldt, O., Juergensen, H., eds.: 4th International Workshop on Implementing Automata. Springer Lecture Notes in Computer Science (2000)
13. Kanthak, S., Ney, H.: Fsa: An efficient and flexible c++ toolkit for finite state automata using on-demand computation. In: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004), Barcelona (2004) 510–517
14. Graehl, J.: Carmel finite-state toolkit. http://www.isi.edu/licensed-sw/carmel (1997)
15. Kaiser, E., Schalkwyk, J.: Building a robust, skipping parser within the at&t fsm toolkit. Technical report, Center for Human Computer Communication, Oregon Graduate Institute of Science and Technology (2001)
16. van Noord, G.: Treatment of epsilon moves in subset construction. Computational Linguistics **26**(1) (2000)
17. Koehn, P., Knight, K.: Feature-rich statistical translation of noun phrases. In: ACL 2003 Proceedings. (2003)

18. Pereira, F., Riley, M.: Speech recognition by composition of weighted finite automata. In Roche, E., Schabes, Y., eds.: Finite-State Language Processing. MIT Press, Cambridge, MA (1997) 431–453
19. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics **23**(2) (1997)
20. Rounds, W.C.: Mappings and grammars on trees. Mathematical Systems Theory **4** (1970) 257–287
21. Och, F.J., Tillmann, C., Ney, H.: Improved alignment models for statistical machine translation. In: Proc. of the Joint Conf. of Empirical Methods in Natural Language Processing and Very Large Corpora, College Park, MD, University of Maryland (1999) 20–28
22. Yamada, K., Knight, K.: A syntax-based statistical translation model. In: ACL Proceedings. (2001) 523–530
23. Eisner, J.: Learning non-isomorphic tree mappings for machine translation. In: ACL Proceedings (companion volume). (2003) 205–208
24. Knight, K., Marcu, D.: Summarization beyond sentence extraction: A probabilistic approach to sentence compression. Artificial Intelligence **139** (2002)
25. Pang, B., Knight, K., Marcu, D.: Syntax-based alignment of multiple translations extracting paraphrases and generating new sentences. In: Proceedings of NAACL. (2003)
26. Charniak, E.: Immediate-head parsing for language models. In: Proceedings of ACL. (2001)
27. Yamada, K.: A Syntax-Based Translation Model. PhD thesis, University of Southern California (2002)
28. Allauzen, C., Mohri, M., Roark, B.: A general weighted grammar library. In: Proceedings of the Ninth International Conference on Automata (CIAA-2004). Volume 3317 of Lecture Notes in Computer Science., Berlin-NY, Springer-Verlag (2004) 23–34
29. Knight, K., Graehl, J.: An overview of probabilistic tree transducers for natural language processing. In: Proceedings of the Sixth International Conference on Intelligent Text Processing and Computational Linguistics. Lecture Notes in Computer Science, Springer Verlag (2005)
30. Thatcher, J.W.: Generalized sequential machines. Journal of Comput. and Syst. Sci. **4** (1970) 339–367
31. Gécseg, F., Steinby, M.: Tree Automata. Akadémiai Kiadó, Budapest (1984)
32. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata` (1997) release October, 1rst 2002.
33. Genet, T., Tong, V.V.T.: Reachability analysis of term rewriting systems with timbuk. In: Proceedings 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. Volume 2250 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2001)
34. Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P., Vittek, M.: Elan: A logical framework based on computational systems. In Meseguer, J., ed.: Proceedings of the first international workshop on rewriting logic, Asilomar (1996)
35. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019. (1995)
36. Huang, L., Chiang, D.: Better k-best parsing. In: Proceedings of IWPT. (2005)
37. May, J., Knight, K.: A better n-best list: Practical determinization of weighted finite tree automata. In: NAACL Proceedings, New York (2006) To Appear.
38. Siztus, A., Ortmanns, S.: High quality word graphs using forward-backward pruning. In: Proceedings of the IEEE Conference on Acoustic, Speech and Signal Processing. (1999) 593–596
39. Lari, K., Young, S.J.: The estimation of stochastic context-free grammars using the inside-outside algorithm. Computer Speech and Language **4** (1990) 35–56
40. Graehl, J.: Context-free algorithms. Unpublished handout (2005)
41. Shieber, S.M.: Synchronous grammars as tree transducers. In: Proceedings of the Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+ 7), Vancouver (2004)
42. Schabes, Y.: Mathematical and Computational Aspects of Lexicalized Grammars. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (1990)
43. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society, Series B **39**(1) (1977) 1–38
44. Graehl, J., Knight, K.: Training tree transducers. In Dumais, S., Marcu, D., Roukos, S., eds.: HLT-NAACL 2004: Main Proceedings, Boston, Association for Computational Linguistics (2004) 105–112
45. Graehl, J., Knight, K., May, J.: Training tree transducers. Computational Linguistics (To Appear)
46. Knight, K., Graehl, J.: Machine transliteration. Computational Linguistics **24**(4) (1998)