Stochastic tree-adjoining grammars for modeling retrotransposons

Lawrence Uricchio, Ian Holmes

Abstract

TAGs and parsers for biological repeats.

Contents

1 Introduction

Transposable elements (TEs), or *transposons*, are of great interest in molecular evolution [1], and an important aspect of genome annotation. There are several specializations in the overall task of transposon annotation: PILER [2] specializes in *de novo* transposon discovery, while REPCLASS specializes in classification of found transposons [3].

Many such programs classify TEs by their general structural features, particularly their terminal repeats: LTRs (Long Terminal Repeats) and TIRs (Terminal Inverted Repeats).

It is useful to build databases and profiles of known transposon families, for the purpose of classifying new ones. To date, the most comprehensive database of known transposons is REPBASE [4], whose profiles rely only on primary sequence homology models; that is, they do not make explicit use of terminal repeat structure.

A promising approach, that combines profile Hidden Markov Models (HMMs) of primary sequence homology (at the level of TE protein domains) with fast algorithms for detecting LTRs, is taken by LTRdigest [5]. The purpose of this paper is to represent the hybrid modeling approach of LTRdigest using formal grammars.

2 Definitions

2.1 Tree-Adjoining Grammars

We define a minimal normal form of Tree-Adjoining Grammars (TAGs) suited to biological sequence analysis, as opposed to the linguistic representation elsewhere [6]. TAGs have previously been used in bioinformatics to model pseudoknots and other RNA structures [7, 8] and to model local duplications (Hickey

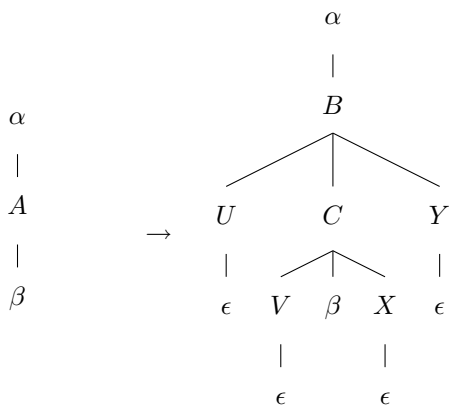
and Blanchette, pers. comm.).

A TAG is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{T}, S, \mathcal{R}, \mathcal{W})$ where $\mathcal{N}$ is a finite set of *node labels*, $\mathcal{T}$ is a finite set of *terminals* (disjoint from $\mathcal{N}$), $S \in \mathcal{N}$ is a distinguished *start label*, $\mathcal{R}$ is a finite set of *transformation rules* and $\mathcal{W} : \mathcal{R} \rightarrow [0, \infty)$ is a *rule weight function*.

The process of generating an output sequence $Z \in \mathcal{T}^*$ using $\mathcal{G}$ is referred to as a *derivation* of Z. The derivation consists of repeated local application of transformation rules to *intermediate trees*. We can commence a derivation from any *initial tree*, but ultimately we will be interested in derivations that commence with the following initial tree

$$\begin{array}{c} \epsilon \\ | \\ S \\ | \\ \epsilon \end{array}$$

Each "intermediate tree" is, formally, an ordered tree whose nodes are labeled from $(\mathcal{N} \cup \mathcal{T}^*)$. The transformation rules can take the various forms shown in Table 1. All the rules may be recognized as special cases of the general form



where $U, V, X, Y \in (\mathcal{N} \cup \mathcal{T}^*)$ and $B, C \in (\mathcal{N} \cup \{\epsilon\}^*)$. The tree is ordered left-to-right.

The derivation stops when no further transformations can be applied. The trees generated by this process have the property that every leaf node is labeled with a terminal sequence, while every internal node is labeled either with ϵ, or with a member of $\mathcal{N}$ which never appears on the left-hand side of a rule in $\mathcal{R}$ (and to which no transformations can therefore be applied).

To obtain the final output sequence, we read off the terminal sequences (at the leaves) in a depth-first traversal of the ordered tree (i.e. from just left of the root, moving anticlockwise around the tree, to just right of the root). The depth-first order of leaves in the general tree shown above is (U, V, β, X, Y).

The *weight* of a derivation is the product of the weights of all rules used in the derivation. The weight of a given output sequence Z is the sum of the weights of all possible derivations of Z. Let $\mathcal{W}[\alpha \Rightarrow Z]$ denote the weight of output sequence Z, starting from initial tree α.

The final column of Table 1 shows a shorthand representation of each rule in

Newick format, probably the most widely-understood bioinformatics format for representing tree structures. The initial tree, in this representation, is $((\epsilon)S)\epsilon$. In the table, we have omitted the placeholder ϵ's at leaf nodes, so that (for example) the rule

$$((\beta)A)\alpha \rightarrow (U, ((v, \beta, x)C), y)\alpha$$

should strictly be read as

$$((\beta)A)\alpha \rightarrow ((\epsilon)U, ((v, \beta, x)C), y)\alpha$$

Let $\mathcal{R}_n(A)$ denote the subset of rules of type n (according to Table 1) where the left-hand side is $((\beta)A)\alpha$ for a given $A \in \mathcal{N}$.

Note the special case of a type-6 rule where all the terminal strings are empty, $((\beta)A)\alpha \rightarrow ((\beta)C)\alpha$. Such a rule is referred to as a *transition* and may be written more compactly as $A \rightarrow C$.

2.2 A parsing algorithm

We can define a general parsing algorithm for TAGs that is the equivalent of the CYK (Cocke-Younger-Kasami) algorithm for SCFGs. The CYK algorithm is closely related to the Inside algorithm, which computes the weight (probability) of a given output sequence. Here, we present the Inside version of the TAG algorithm; to obtain the CYK version, simply replace all summation operators (Σ) with max operators.

For the given output sequence $Z \in \mathcal{T}^*$, let $Z[i \ldots j+1]$ denote the substring from i through j inclusive, for $1 \leq i \leq j \leq |Z|$. Let $Z[i \ldots i] = \epsilon$.

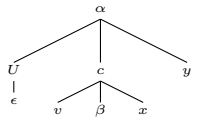
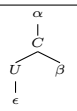
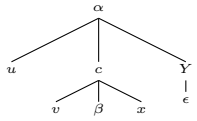
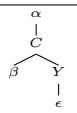
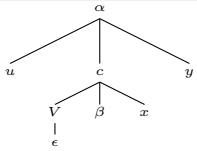
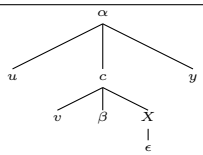
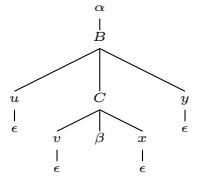
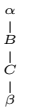
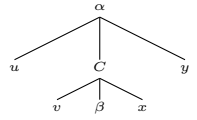
Type	From		To	Newick representation	When $uvxy = \epsilon$:
(1)		→		$((\beta)A)\alpha \rightarrow (U, ((v, \beta, x)C), y)\alpha$	
(2)		→		$((\beta)A)\alpha \rightarrow (u, ((v, \beta, x)C), Y)\alpha$	
(3)		→		$((\beta)A)\alpha \rightarrow (u, ((V, \beta, x)C), y)\alpha$	
(4)		→		$((\beta)A)\alpha \rightarrow (u, ((v, \beta, X)C), y)\alpha$	
(5)		→		$((\beta)A)\alpha \rightarrow ((u, ((v, \beta, x)C), y)B)\alpha$	
(6)		→		$((\beta)A)\alpha \rightarrow (u, ((v, \beta, x)C), y)\alpha$	

Table 1: Types of transformation rule (i.e. tree adjunction rule) used in this paper. Here α, β represent any subtree; $A \in \mathcal{N}$ is the source node label; $B, C, U, V, X, Y \in (\mathcal{N} \cup \{\epsilon\})$ are the destination node labels; ϵ is the empty string; and $u, v, x, y \in \mathcal{T}^*$ are (possibly empty) terminal strings. The final column shows the destination tree when all the terminal strings u, v, x, y are empty.

Define some indicator functions to match output substrings to rules

$$\begin{aligned}
\Delta(i,j,x) &= \delta\left(Z[i\ldots j] \;=\; x\right) \\
\overrightarrow{\Delta}\,(i,x) &= \Delta(i,i+|x|,x) \\
\overleftarrow{\Delta}\,(j,y) &= \Delta(j-|y|,j,y) \\
\Delta_{iu} &= \overrightarrow{\Delta}\,(i,u) \\
\Delta_{jv} &= \overleftarrow{\Delta}\,(j,v) \\
\Delta_{kx} &= \overrightarrow{\Delta}\,(k,x) \\
\Delta_{ly} &= \overleftarrow{\Delta}\,(l,y)
\end{aligned}$$

2.2.1 The dynamic programming matrix

Introducing the placeholder γ as an additional terminal, define

$$\begin{aligned}
M(i,j,k,l,A) &= \mathcal{W}\left[((\gamma)A)\epsilon \Rightarrow Z[i\ldots j]\gamma Z[k\ldots l]\right] \\
M'(i,l,A) &= \mathcal{W}\left[((\epsilon)A)\epsilon \Rightarrow Z[i\ldots l]\right]
\end{aligned}$$

where $1 \le i \le j \le k \le l \le |Z|$.

Thus $M(i,j,k,l,A)$ is the probability that the initial tree $((\gamma)A)\epsilon$ will generate $Z[i\ldots j]$ to the left of γ and $Z[k\ldots l]$ to the right of γ, whereas $M'(i,l,A)$ is the probability that the initial tree $((\epsilon)A)\epsilon$ will generate $Z[i\ldots l]$. Note that

$$M'(i,l,A) = \sum_{k=i}^{l} M(i,k,k,l,A)$$

Note also the following boundary conditions:

$$\begin{aligned}
M(i,i,k,k,\epsilon) &= 1 \\
M'(i,i,\epsilon) &= 1
\end{aligned}$$

The *inside sequence* refers to the sequences $Z[i \dots j]$ and $Z[k \dots l]$ (for M), or the sequence $Z[i \dots l]$ (for M').

2.2.2 The dynamic programming recursion

The DP recursion relation is as follows (with $B, C, U, V, X, Y, u, v, x, y$ defined as in Table 1)

$$
\begin{aligned}
M(i,j,k,l,A) = & \\
& \sum_{\rho \in \mathcal{R}_1(A)} \sum_{m=i}^{j} W(\rho) M(m, j-|v|, k+|x|, l-|y|, C) M'(i,m,U) \Delta_{jv} \Delta_{kx} \Delta_{ly} \\
& + \sum_{\rho \in \mathcal{R}_2(A)} \sum_{n=k}^{l} W(\rho) M(i+|u|, j-|v|, k+|x|, n, C) M'(n,l,Y) \Delta_{iu} \Delta_{jv} \Delta_{kx} \\
& + \sum_{\rho \in \mathcal{R}_3(A)} \sum_{m=i}^{j} W(\rho) M(i+|u|, m, k+|x|, l-|y|, C) M'(m,j,V) \Delta_{iu} \Delta_{kx} \Delta_{ly} \\
& + \sum_{\rho \in \mathcal{R}_4(A)} \sum_{n=k}^{l} W(\rho) M(i+|u|, j-|v|, n, l-|y|, C) M'(k,n,X) \Delta_{iu} \Delta_{jv} \Delta_{ly} \\
& + \sum_{\rho \in \mathcal{R}_5(A)} \sum_{m=i}^{j} \sum_{n=k}^{l} W(\rho) M(i+|u|, m, n, l-|y|, B) M(m, j-|v|, k+|x|, n, C) \Delta_{iu} \Delta_{jv} \Delta_{kx} \Delta_{ly} \\
& + \sum_{\rho \in \mathcal{R}_6(A)} W(\rho) M(i+|u|, j-|v|, k+|x|, l-|y|, C) \Delta_{iu} \Delta_{jv} \Delta_{kx} \Delta_{ly}
\end{aligned}
$$

Note that for M to be exactly computable, we require that there are no *null cycles* in the grammar. A null cycle is a series of transformations that, when applied consecutively to a given tree α, yield the original tree α again, or one that is trivially related to it (e.g. a tree that is identical to α after ϵ-labeled nodes of degree < 3 have been removed). Null cycles frequently arise via consecutive transitions of the form $A \to B$ followed by $B \to A$, but there are other possibilities too (for example, a type-1 bifurcation rule $((\beta)A)\alpha \to ((U,\beta)C)\alpha$ followed by two transitions $U \to \epsilon$ and $C \to A$).

A *Lexicalized TAG* (LTAG) has the property that one of u, v, x, y is always a nonempty terminal string, guaranteeing that it contains no null cycles, and ensuring that every rule explicitly matches some output characters. LTAGs are popular in linguistics, though the constraint is possibly too stringent to be helpful for bioinformatics grammars.

2.2.3 Fill order

The absence of null cycles implies that we can perform a *topological sort* (topo-sort) of $\mathcal{N}$ based on the transition graph, yielding the order in which node labels must be visited in the Inside algorithm.

A suitable (but non-unique) Inside fill order is as follows:

- For $L_1 = 0$ to $|Z|$ (increasing), $\quad i = 0$ to $|Z| - L_1$ (increasing):
  - Let $l = i + L_1$
  - For $L_2 = L_1$ to 0 (decreasing), $\quad j = i$ to $i + L_1 - L_2$ (increasing):
    - Let $k = j + L_2$
    - For $A \in \mathcal{N}$ (topo-sorted, transition destinations before sources):
      - Compute $M(i, j, k, l, A)$ and store
  - For $A \in \mathcal{N}$ (topo-sorted, transition destinations before sources):
    - Compute $M'(i, l, A)$ and store

2.2.4 Length distributions

To model genomic features, it is extremely useful to augment the TAG framework with length distributions; that is, to allow the rule weight $\mathcal{W}(\rho)$ to depend on distances such as $m - i$, $j - m$, $n - k$, $l - n$, $n - m$, $j - i$, $k - j$, $l - k$, $l - i$ and so on.

Note that the TAG framework implicitly allows some length distributions to be constructed, even without such augmentation; for example, the "waiting time" in a particular state is geometrically-distributed, and by arranging a series of such states, one can obtain other distributions, e.g. a negative binomial distribution. One can also obtain any distribution over a random variable $x \leq N$ by using N states, or approximate any distribution to some degree of precision with a finite number of states. However, it is often much more convenient and efficient simply to allow $\mathcal{W}$ to sometimes depend directly on the inside sequence length.

3 A simple retrotransposon grammar

We are particularly interested in following class of grammars that describe specific arrangements of DNA-encoded protein domains flanked by LTRs (long terminal repeats).

3.1 SCFG and LTR components

The LTRs can be generated by type-6 rules (e.g. with $v = y = \epsilon$) followed either by a transition to the transposon interior, or (optionally) a type-5 adjunction rule, which may be an easier event on which to impose a hint-requirement constraint.

3.2 Developing the SCFG sub-grammar for transposon contents

- X generates a nucleotide sampled from the background distribution
- X_L generates ℓ background nucleotides, where $\ell \sim L$
- F_N samples a DNA sequence coding for family N from PFAM [9]

- I_L generates an intron of length $\ell \sim L$ (can be emitted by F_N)
- T_A generates a terminal inverted repeat, then transits to A

We can also make transitions back to S to generate a nested transposon insertion.

The grammar so described has some similarities to LTRdigest [5] and TENest [10].

3.3 Supplying external hints

The parsing algorithm uses $\mathcal{O}(|Z|^4)$ memory and $\mathcal{O}(|Z|^6)$ time. The hope is to accelerate it significantly by using externally-supplied "hints" as constraints on the locations of various features (especially the LTRs).

The hints file should include

- A set of tuples (i, j, k, l) indicating that $Z[i \dots j]$ and $Z[k \dots l]$ are (respectively) the 5' and 3' repeat regions of an LTR
- A set of tuples (i, j, N) indicating that $Z[i \dots j]$ is a match to PFAM family N

These hints can be generated by fast tools, e.g. suffix-tree based algorithms for finding LTRs [11], or GeneWise for finding DNA sequences that code for PFAM protein domains [12].

A very quick heuristic that might achieve most of the benefits of a more rigorous "hints" constraint would be to divide the genome into windows and only run the grammar on windows which contain K or more of the appropriate hints.

4 Glossary of mathematical notation

Symbol	Meaning
$\mathcal{G}$	Grammar
$\mathcal{N}$	Set of node labels
$\mathcal{T}$	Set of terminals
$\mathcal{R}$	Set of transformation rules
$\mathcal{R}_n(A)$	Set of transformation rules of type n (see Table 1) whose LHS is $((\beta)A)\alpha$
α, β	Arbitrary ordered node-labeled trees
$\mathcal{W}$	Rule weight function
$\mathcal{W}[\alpha \Rightarrow Z]$	Sum, over derivations of Z from α, of product over derivation rule weights
ϵ	The empty string
$\mathcal{T}^*$	Set of strings over $\mathcal{T}$, including the empty string
Z	Output sequence
$\|Z\|$	Length of Z
$Z[i \dots j+1]$	Substring of Z from i to j inclusive (i starts at 1)
$Z[i \dots i]$	The empty string
$\Delta(i, j, x)$	Indicates if rule string x matches output string $Z[i \dots j]$
$\overrightarrow{\Delta}(i, x)$	Indicates if rule string x matches output string Z, starting at position i
$\overleftarrow{\Delta}(j, y)$	Indicates if rule string y matches output string Z, ending before position j
$\Delta_{iu}, \Delta_{jv}, \Delta_{kx}, \Delta_{ly}$	Rule string matchers; Δ_{ns} matches string s ending/starting at position n
$M(i, j, k, l, A)$	Inside weight for $Z[i \dots j]$ (left) and $Z[k \dots l]$ (right) rooted at A
$M'(i, l, A)$	Inside weight for $Z[i \dots l]$ rooted at A
B, C, U, V, X, Y	Labels (or terminal strings) at nodes of the Newick tree $((U, ((V, \beta, W)C), X)B)\alpha$
u, v, x, y	Special symbols for U, V, X, Y when they are terminal strings

5 Acknowledgments

Sean Eddy co-organized an excellent workshop at U.Penn, "Language Modeling of Biological Data", at which IH met Mark Steedman, Bonnie Webber and Aravind Joshi, and learned from them (and others) the secrets of TAGs, finite-state transducers, and mildly context-sensitive grammars. David Searls has been a long-time proponent of linguistics applied to bioinformatics. Dan Klein encouraged us to develop our own variations on grammar frameworks. Michael Souza introduced us to the pumping lemma and the equivalence of TAGs with various other frameworks.

6 References

References

1. C. Feschotte. DNA transposons and the evolution of eukaryotic genomes. *Annual Review of Genetics*, 41:331–368, 2007.

2. R. C. Edgar and E. W. Myers. PILER: identification and classification of genomic repeats. *Bioinformatics*, 21 Suppl 1:i152–8.

3. C. Feschotte, U. Keswani, N. Ranganathan, M. L. Guibotsy, and D. Levine. Exploring repetitive DNA landscapes using REPCLASS, a tool that automates the classification of transposable elements in eukaryotic genomes. *Genome Biol Evol*, 1:205–220, 2009.

4. V. V. Kapitonov and J. Jurka. A universal classification of eukaryotic transposable elements implemented in Repbase. *Nature reviews. Genetics*, 9(5):411–2; author reply 414.

5. S. Steinbiss, U. Willhoeft, G. Gremme, and S. Kurtz. Fine-grained annotation and classification of de novo predicted LTR retrotransposons. *Nucleic Acids Res.*, 37:7002–7013, Nov 2009.

6. A. Joshi and Y. Schabes. Tree-adjoining grammars, 1997.

7. H. Matsui, K. Sato, and Y. Sakakibara. Pair stochastic tree adjoining grammars for aligning and predicting pseudoknot RNA structures. *Bioinformatics*, 21:2611–2617, Jun 2005.

8. D. Chiang, A. K. Joshi, and D. B. Searls. Grammatical representations of macromolecular structure. *J. Comput. Biol.*, 13:1077–1100, Jun 2006.

9. R. D. Finn, J. Tate, J. Mistry, P. C. Coggill, S. J. Sammut, Hans-Rudolf Hotz, G. Ceric, K. Forslund, S. R. Eddy, E. L. L. Sonnhammer, and A. Bateman. The Pfam protein families database. *Nucleic Acids Research*, 36(Database issue):D281–8, 2008.

10. B. A. Kronmiller and R. P. Wise. TEnest: automated chronological annotation and visualization of nested plant transposable elements. *Plant Physiol.*, 146:45–59, Jan 2008.

11. A. Kalyanaraman and S. Aluru. Efficient algorithms and software for detection of full-length LTR retrotransposons. *J Bioinform Comput Biol*, 4:197–216, Apr 2006.

12. E. Birney, M. Clamp, and R. Durbin. GeneWise and GenomeWise. *Genome Research*, 14(5):988–995, 2004.