

Learning a Subclass of Tree Adjoining Grammars

Sahar Al Seesi*, Sanguthevar Rajasekaran, Reda Ammar
Computer Science and Engineering Department, University of Connecticut

Abstract

Researchers have defined Tree Adjoining Grammars with tags (TAG^2) [12][20] as an extension to Tree Adjoining Grammars (TAG) [9]. They have used their model to represent RNA secondary structures with crossing dependencies. In this paper, we define a subclass of TAG^2 , Linked Single Adjoining-Tree Adjoining Grammars with tag (LSA- TAG^2). We have developed an inference algorithm that can identify the defined subclass from positive structural examples. The algorithm uses an oracle that can answer subset and equivalence queries. LSA- TAG^2 s are capable of modeling RNA structures with crossing dependencies.

Keywords: Grammatical Inference, Formal Grammars, TAG, RNA modeling.

*Contact author
email: sahar@engr.uconn.edu
Computer Science & Engineering Department,
University of Connecticut
371 Fairfield Road
Unit 2155
Storrs, CT 06269-2155
USA
phone : 1-860-730-2763
fax: : 1-860-486-4817

1. Introduction

Grammatical inference is the problem of learning or identifying the grammar of an unknown language given a finite sample of this language. We consider the problem of learning a subclass of Tree Adjoining Languages (TALs) with links from examples and queries. Tree Adjoining Grammars (TAGs) were introduced by Joshi *et. al.* [9] for use in the field of natural language processing. In TAGs with links [8], the elementary trees, which are the building blocks for a TAG, can have links that capture dependencies between nodes. Kobayashi and Yokomori [12] defined TAG² as an extension of TAG where some internal nodes can be tagged with the symbol ‘*’. Adjoining, which is the main composition operation in TAGs, can only occur at tagged nodes. They defined a generic grammar TAG²_{RNA} to model RNA sequences in [12] and in a follow up paper by Uemura *et. al* [20]. TALs are a superset of Context Free (CF) languages and a subset of Context Sensitive (CS) languages. Other works in field of learning languages higher than CF includes [3] and [4].

One of the oldest established models for grammatical inference is Gold’s identification in the limit[7]. In his model, the learning algorithm accepts an infinite sample of the unknown language, L' , one example at a time. At every point in time, the algorithm makes a guess of the grammar of the unknown language. A class of languages is identifiable in the limit if there exists an algorithm which, for every language L' in this class, will converge in a finite time to a grammar G such that $L(G) = L'$. The sample can either be a positive sample S^+ or a complete sample (S^+, S^-) , where $S^+ \subseteq L'$ and $S^- \subseteq L'$. In [17], Sakakibara made use of the structure of the input examples to learn CF languages.

Another model of inference is learning through queries. This model was introduced by Angluin who studied different types of queries and their application in algorithmic learning [1]. Subset and equivalence queries are two types of queries she defined. A subset query accepts a grammar G . It returns true if $L(G) \subseteq L'$ and false otherwise. An equivalence query accepts a grammar G . It returns true if $L(G) = L'$ and false otherwise. Other types of queries include membership and superset queries. Angluin used membership and equivalence queries in learning regular grammars in [2]. As an extension to her work, Sakakibara [18][15] used structural membership and equivalence queries in learning context free grammars. Queries were also used in the identification of pattern languages [11]. A good survey of the different inference models and the current research under each model can be found in [17].

In this paper, we define a subclass of TAG² called Linked Single Adjoining (LSA)-TAG², and we present an algorithm that identifies the defined subclass given a positive sample of derived tree skeletons with links for the unknown grammar. Our algorithm differs from Gold’s model in that it has to process the whole input sample before inferring any grammar. A skeleton is a derived tree whose internal nodes are not labeled. The skeletons are used to provide information about the structure of the input examples. The algorithm also makes use of an oracle that can answer subset and equivalence queries.

The algorithm decomposes the input skeletons into a set of elementary trees that will accept the input sample. The generated grammar is then generalized by reducing trees with matching skeletons under certain restrictions. The reduction process is similar in essence to what Fu and Booth [6] described for deriving regular grammars from canonical definite grammars by partitioning their set of non-terminals. The same idea was modified and used in [14] to identify regular languages in the limit. Our motivation for developing a grammatical inference algorithm for this class of languages lies in their capability to deal with crossing dependencies and consequently model pseudoknotted RNA structures.

We start this paper by defining the grammar model we are dealing with in section 2. Section 3 introduces some preliminaries and basic definitions that will be used throughout the paper. Section 4 presents the algorithm in detail. The identification proof and the complexity analysis are presented in sections 5 and 6, respectively. Finally, we conclude in section 7 by summarizing the results and providing a list of open problems and future research directions.

2. Model Definition

A Tree Adjoining Grammar (TAG), as defined in [8], is a 5-tuple (T, N, I, A, S) , where T , N , and S are: the set of terminals, the set of non-terminals, and the starting symbol respectively. I and A are defined as follows:

I (initial trees): A finite set of finite trees with the internal nodes' labels belonging to N , the leaves' labels belonging to T , and the root being labeled with S .

A (auxiliary trees): A finite set of finite trees with the internal nodes' labels belonging to N , and the leaves' labels belonging to T except one leaf node which is labeled by the same non-terminal as the root. This special leaf node is called a foot node.

We will refer to leaves with terminal labels as terminal nodes, and we will refer to internal nodes and the foot node as non-terminal nodes.

Trees belonging to $I \cup A$ are called elementary trees. A tree derived by composing two other trees is called a derived tree. Trees can be composed together using the adjoining operation¹. The adjoining operation composes an auxiliary tree α with a foot node labeled X with any other tree β that has some internal node with the same label X . The operation works as follows: we start with the tree β and we extract the sub-tree rooted at the internal node labeled with X (let that sub-tree be γ), and replace it with α . Then at the foot node of α , we reinsert γ . The adjoining operation is illustrated in Figure 1. Let $T = \{ t : \exists i \in I \text{ s.t. } t \text{ can be derived from } i \}$, then $L(\text{TAG})$ would consist of the yield of all the trees in T . In TAG with links, dependencies between elementary tree nodes are defined through links. These links get stretched during derivation to maintain such dependencies. Figure 2 depicts links in an auxiliary tree and how they get stretched in a derived tree.

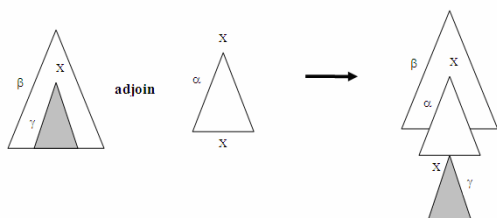


Figure 1 The adjoining operation

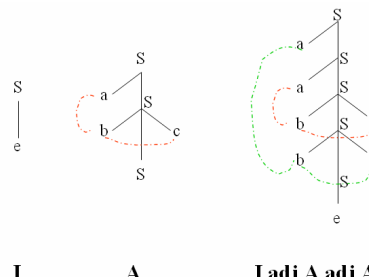


Figure 2 TAG with links

In [12], TAG^2 is defined to be an extension of TAG where some internal nodes can be tagged with the symbol '*'. Adjoining can only occur at tagged nodes. Their extension is similar to TAG with null adjoining (NA) constraint, as defined in [10], applied to all the internal nodes except those tagged with the special symbol '*'. In addition, a set of final symbols $F \subseteq N$ is defined. The language defined by TAG^2 consists of the yield of all derived trees where the labels of tagged non-terminal nodes are in F .

Definition: An LSA- TAG^2 is a TAG^2 with links and the following additional restrictions:

- 1) Initial trees have only two levels: root (which is the adjoining node), and leaves
- 2) All internal nodes in auxiliary trees appear only on the backbone of the tree. The backbone is the path between the root and the foot node
- 3) Exactly one node per tree is tagged (labeled) for adjoining; thus the prefix Single Adjoining.
- 4) All terminal leaves in an elementary tree are linked together
- 5) No unnecessary nodes: This means every internal node must have terminal node children except under one of two conditions. A node can have only one child which is a non-terminal node only if:

¹ There is another definition of TAG [9][10] where initial trees can have non-terminals in the leaves and auxiliary trees can have more than one non-terminal in the leaves. In that model there is another composition operation called substitution.

- a. The first is the root and the second is tagged for adjoining and has a different non-terminal label than the root.
- b. The first is tagged for adjoining and the second is the foot node and they have different non-terminal labels.

In all the other cases, the two nodes can be collapsed into one without affecting the language represented by the grammar.

The main two limiting restrictions of the above are restrictions (2) and (3). A detailed study of the properties of LSA-TAG² will be the focus of a future publication. It suffices to say for now that TYPE1-TYPE4 trees of TAG²_{RNA} [12], can be represented with slight modifications under LSA-TAG². However, the model does not cover TYPE5 trees.

From this point forward, we will refer to an LSA-TAG² with the list of trees that defines it. To differentiate between terminals and non-terminals we will use upper case to denote a non-terminal and lower case to denote a terminal. The starting symbol is always S.

3. Preliminaries:

A **tree** T is defined as a set of nodes. Each node, d, is a 5-tuple (p, ch, lnks, lb, tag), where p is the parent node of d, ch is a set of children nodes of d, lnks is defined for terminal nodes and it is the set of leaf nodes linked to d as defined in TAGs with links [8] (Figure 2 illustrates links), lb is the terminal or non-terminal label of the node, and tag, defined for non-terminal nodes, is a Boolean specifying whether the node is tagged for adjoining or not. Define mappings **root**(T), **footnode**(T), **adjoining_node**(T) which map a tree, T, to its root, foot node or the adjoining node, respectively, if there is one. For a node d, we define **label**(d), **links**(d), **tag**(d), **parent**(d), and **children**(d) which return the corresponding data for the node as indicated by the mapping name. We also define **level**(T,d) which returns the depth of a node d in a tree T.

Define a **skeleton** to be a tree where all non-terminal nodes have label = *undefined*. Define an **incomplete tree** to be a tree where the labels for non-terminal nodes $\in NT \cup \{undefined\}$. The rank of a non-terminal, **rank**(nt), is defined to be the order in which it was first used in the inference process relative to other non-terminals. **skeleton**(T) is a mapping from a tree or an incomplete tree, T, to its skeleton. **tagged_skeleton**(T) is a mapping from a tree or an incomplete tree, T, to its skeleton while preserving the tag on the adjoining node. We will be loosely using the term tree to refer to incomplete trees. Given a set of trees $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$, **SK**(\mathcal{T}) = {skeleton(T₁), skeleton(T₂), ... skeleton(T_n)}. **TSK** is similarly defined for tagged_skeleton. A **primitive** tree is defined to be a tree where all the terminal nodes are linked together.

The following three tree operations are defined. **Extract**(T,A,B) extracts from T the tree section bound by internal nodes A and B. **Extract**(T,A,B) = {d: d \in T and level(T,A) < level(T,d) \leq level(T,B) and d \neq B} \cup {(NULL, ch, lnks, tag, lb) : A = (parent, ch, lnks, tag, lb)} \cup {(p, \emptyset , lnks, tag, lb) : B = (p, ch, lnks, tag, lb)}. The second operation is the subtraction operation '-'. It is used to calculate the resulting tree after extracting a T₂ from T₁. T₁ - T₂ = {d = (p, ch, lnks, tag, lb): d \in T₁ and (d \notin T₂ or root(T₂) \neq (NULL, ch, lnks, tag, lb))}. The resulting tree, T, will have a broken link. This means there exist two nodes A and B in T such that A has a child C where C \notin T and parent(B) \notin T. Therefore, we define the third operation **AdjustBrokenLinks**(T). AdjustBrokenLinks removes C from children(A) and adds B instead. It also sets parent(B) to be equal to A.

A **structurally complete sample** for a TAG is a sample in which each possible adjoining operation is represented in the sample.

4. The Inference Algorithm

Our inference algorithm accepts as an input a set of positive derived trees' skeletons with links. The algorithm is a decompose/reduce/verify algorithm. It makes use of subset and equivalence queries during its various phases. The decompose phase generates one or more grammars. Each one is a definite

grammar which only accepts the set of positive examples. Among the generated grammars, there is at least one solution G where $SK(G) = SK(G')$; assuming the unknown language $L' = L(G')$. After the decomposition is done, each generated definite grammar is reduced/generalized to the smallest/most general set of trees that accepts the positive examples and at the same time does not accept any string in L' . This is ensured by executing a subset query after every reduction step. At the third phase of the algorithm, we use equivalence queries to pick a correct solution. As presented here, the algorithm does not generate the set of final non-terminals F . However, this can be handled with slight modifications.

Before explaining our decomposition logic, let us first study the possible adjoining options allowed in LSA-TAG²s. An auxiliary tree can have one adjoining node. This adjoining node can either be the root, the foot node, or an internal node. If the adjoining node has no terminal children, then we know it is the child of the root or the parent of the foot node. Otherwise, it is considered an unnecessary node which is not allowed in LSA-TAG². Figure 3 shows the results of adjoining the shaded tree, at the adjoining node, tagged with '*', for all possible positions of the adjoining node. Type d and type e represent the two special cases of an adjoining node having no terminal node children.

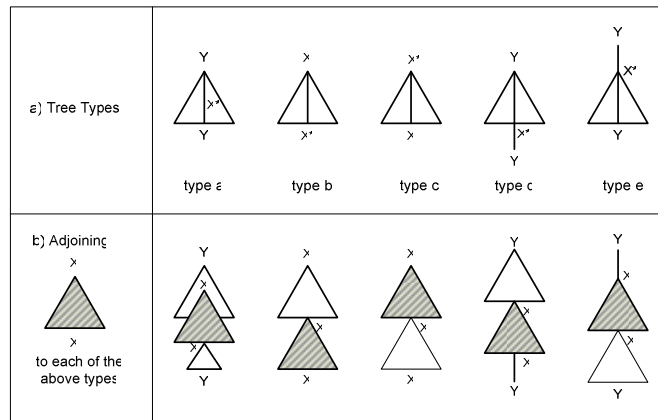


Figure 3 Adjoining nodes options

4.1 The Decompose Phase

The decompose phase accepts one positive example at a time. The decomposition process starts by detaching the initial tree from the bottom of the input skeleton (Figure 4 (b)). In LSA-TAG²s, initial trees have only two levels: the root, which is the adjoining node, and the leaves. Thus, the set of lowest linked nodes and their parent can be extracted from the skeleton. The starting symbol S is used to label the parent node, and the extracted tree is added to the set of inferred trees. Once the initial tree is inferred, it is subtracted from the skeleton, and the starting symbol is used to label root and foot node of the resulting tree after adjusting its broken links. If the resulting tree is primitive, it will be added to the set of inferred trees. Otherwise, it undergoes a recursive decomposition process that leads to inferring two or more auxiliary trees.

The decomposition process breaks down a complex tree into two simpler trees by applying the opposite of one of the adjoining options. For each of the auxiliary tree types illustrated in Figure 3 there is a corresponding decomposition function: Outer (type a), High (type b), Low (type c), ExtendedHigh (type d), and ExtendedLow (type e). Each of the five decomposition functions is responsible for performing the following tasks: breaking down the tree into two simpler trees, determining the adjoining node that resulted in this composition and tagging it, and labeling the root and foot node of both trees and the node tagged for adjoining. The decomposition process can result in either two primitive trees or a primitive tree and a complex tree. A resulting complex tree is recursively sent to the decomposition function. A resulting primitive tree is added to the set of inferred trees unless it does not comply with the model. First, we will discuss how the decomposition and tagging tasks are performed. Then, we will discuss the node labeling.

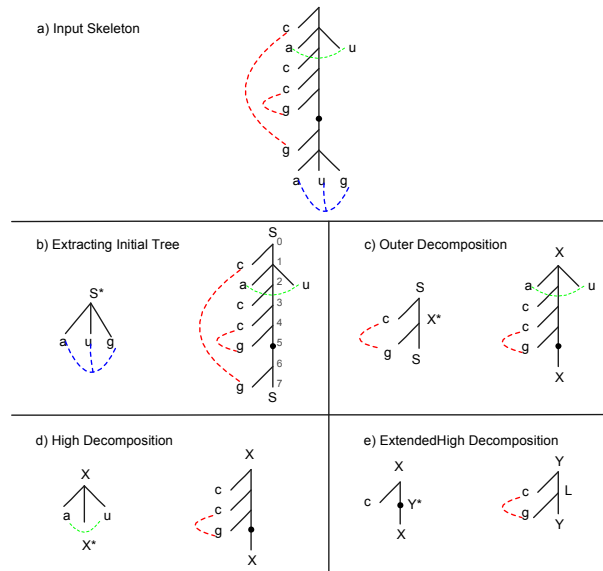


Figure 4 Decomposition Example

4.1.1. Decomposition and Tagging

Choosing a decomposition function depends on evaluating the eight inference conditions in Table 1. Let d_h and d_l be the highest and lowest terminal nodes in a tree T respectively. The root is always at level 0. That makes d_h the terminal node with the lowest level and d_l the terminal node with the highest level. Let i be the level of d_h and j be the highest level of any terminal node linked to d_h . Let q be the level of d_l and p be the lowest level of any terminal node linked to d_l . The depth of $T = n + 1$. Table 2 lists the possible combination values of the inference conditions and the corresponding inference function choice. As obvious in the table, there are conflicts between decomposition functions High and ExtendedHigh, High and Low, and Low and ExtendedLow. In such cases, the decomposition path branches into two paths maintaining two sets of inferred elementary trees. At any point in the inference process, a decomposition path can fail and terminate. A decomposition path fails if it leads to inferring an auxiliary tree that does not comply with the model constraints.

Condition Number	Condition
ic1	$i = 1$
ic2	$j = n$
ic3	$p = 1$
ic4	$q = n$
ic5	$ children(root(T)) = 1$
ic6	$ children(parent(footnode(T))) = 1$
ic7	Tree in between levels $i-1$ and j inclusive of both is primitive
ic8	Tree in between levels $p-1$ and q inclusive of both is primitive

Table 1 Inference Conditions' Codes

4.1.1.1 Outer Decomposition

Outer decomposition is chosen when the highest and lowest terminal leaves are linked together. Because we are inside a decomposition function, we know that T is not a primitive tree. Consequently, there are some terminal nodes that are not a part of this chain of links. This means that if we look at the levels of terminal nodes in this chain, these levels will not cover all the levels in the tree. We refer to each maximal set of consecutive levels not covered by the chain as a gap. In the complex tree shown in Figure 4(b), $n = 7$, d_h is labeled 'c', d_l is labeled 'g', $i = p = 1$, $j = q = n$, and $\{2, 3, 4, 5, 6\}$ is a gap. According to LSA-TAG² definition, there can only be one adjoining node per elementary tree. Therefore, there can

only be one such gap in T. Let $\{a, a+1, \dots, b\}$ be the gap in T. The decomposition process, depicted in Box 1, extracts the tree section, TI, rooted at level a -1 and ending at level b. In the example shown in Figure 4, that would be the tree rooted at level 1 and ending at level 6. The tree, TO, resulting from subtracting TI from T is a primitive tree. We tag for adjoining the node in TO corresponding to where the extraction occurred in T. Figure 4 (c) illustrates the result of applying outer decomposition on the complex tree shown in Figure 4 (b).

Decomposition Function / Auxiliary Tree Type	ic1	ic2	ic3	ic4	ic5	ic6	ic7	ic8
Outer / a	T*	T*	T	T	F	F	F	F
High / b	T*	F*	F	DC	F*	DC	T*	DC
Ext. High / d	T*	F*	F	F	F	T*	T*	DC
Low / c	DC	F	F*	T*	DC	F*	DC	T*
Ext. Low / e	F	F	F*	T*	T*	F	DC	T*

Table 2 Inference Condition Values and Corresponding Decomposition Functions

DC : Don't Care condition; * : sufficient conditions

4.1.1.2 High Decomposition

In high decomposition, we start from the tree top. First, we identify the highest terminal node, d_h , in T and the set of leaves in $\text{Inks}(d_h)$. In the complex tree shown in Figure 4 (c), d_h is labeled 'a', $i = j = 1$. Note that: (1) T is not a primitive tree because we are in a decomposition function; (2) the tree in between levels i-1 and j inclusive of both is primitive because ic7 is a necessary condition for High Decomposition. Thus, we know that there are terminal nodes lower than the nodes in $\text{Inks}(d_h)$. In our example, those are the three nodes labeled 'c', 'c', and 'g'. We decompose T into two trees. The upper, TU, has all the leaves in $\text{Inks}(d_h)$. The lower tree, TL, has all the leaves that are lower than the lowest leaf in $\text{Inks}(d_h)$. Due to the fact that ic7 is true, we know that TU is a primitive tree. The foot node of TU is tagged for adjoining. Box 2 includes the pseudo code for High decomposition. Figure 4 (d) illustrates the result of applying outer decomposition on the complex tree shown in Figure 4 (c).

Outer(T)

- let a be the lowest terminal node in T
- let B = internal node s.t. level (T,B) = i and $C = \langle p_c, ch_c, \phi, lbl_c, tag_c \rangle$, internal node s.t. level (T,C) = j - 1 where 1, 2, ..., i, j, j+1, ..., n are the levels for leaves in links(a) and n+1 is the depth of T
- $TI = \text{Extract}(T, B, C)$
- $TO = T - TI$
- let $C_{to} = \langle p_{to}, ch_c, \phi, lbl_c, tag_c \rangle$, $C_{to} \in TO$
- set $tag(C_{to}) = 1$
- AdjustBrokenLinks(TO)
- adjoining_node_label = AddAuxiliaryTree (TO)
- if failure
- exit; terminate this inference path
- set label(root(TI)) = adjoining_node_label
- set label(footnode(TI)) = adjoining_node_label
- if TI is primitive
- AddAuxiliaryTree(TI)
- if failure
- exit; terminate this inference path
- else
- Decompose(TI)

Box 1 Outer Decomposition

High(T)

- let a be the highest terminal node in T, and b be the lowest node in $\text{Inks}(a)$
- let A and B be the parents of a and b respectively
- let C = non-leaf child of B = $\langle p_c, ch_c, \phi, lbl_c, tag_c \rangle$
- set $TU = \text{extract}(T, A, C)$
- set $TL = T - TU$
- AdjustBrokenLinks(TL)
- LabelEnds(TU)
- LabelEnds(TL)
- let $C_{tu} = \langle p_c, \phi, \phi, lbl_c, tag_c \rangle$ where $C_{tu} \in TU$
- set $tag(C_{tu}) = 1$
- AddAuxiliaryTree (TU)
- if failure
- exit; terminate this inference path
- if TL is primitive
- AddAuxiliaryTree(TL)
- if failure
- exit; terminate this inference path
- else
- Decompose(TL)

Box 2 High Decomposition

4.1.1.3 ExtendedHigh Decomposition

This kind of decomposition is the same as the previous except that it includes in the resulting upper tree TU, the foot node of T. The parent of footnode(TU) is tagged for adjoining. The resulting primitive tree will be of type d. Box 3 includes the pseudo code for ExtendedHigh decomposition. Figure 4 (e) illustrates the result of applying outer decomposition on the complex tree shown in Figure 4 (d).

The decomposition functions Low and ExtendedLow are mirror images of High and ExtendedHigh. These two functions start decomposition from the tree bottom. We will not go into the details of these two functions due to space constraints.

4.1.2 Labeling Nodes

The main idea of node labeling is to establish a link between the node marked for adjoining during the decomposition process and the root and foot node of the other resulting tree. As mentioned before, this step generates definite grammar(s). Every time an adjoining node is determined it is labeled with a new non-terminal label. Thus, the number of non-terminals and consequently the number of resulting distinct trees will be large. It is during the reduction step that the number of primitive trees and non-terminal labels used are reduced to reflect the actual grammar. Note that for any TAG G, $L(G)$ is independent the label of any non-terminal node d, unless d is the root, foot node, or adjoining node. We call these non-critical nodes. Consequently, we will label any non-critical node d with a dummy label 'L'. Define $\text{rank}('L')$ to be ∞ .

Please note that all the decomposition functions call AddAuxiliaryTree. AddAuxiliaryTree(T) is responsible for making sure that T does not violate the model constraints, labeling the non-critical nodes with the dummy label, labeling the adjoining node with a new non-terminal label if it is not already labeled, and adding the tree to the inferred set of trees if it does not already belong to it. AddAuxiliaryTree returns the label of the adjoining node to the calling function.

Decomposition functions vary in the way they use AddPrimitiveTree. Outer, ExtendedHigh and ExtendedLow use it to label and add the tree with the adjoining node to the inferred tree set. Then they use the return value to label the root and foot node of the other tree (TI, TL, or TH respectively).

In High and Low, on the other hand, once the tree has been broken down into two simpler trees, LabelEnds is called to match the labels of the foot node and root of each tree. In this case, AddPrimitiveTree does not label the adjoining node because it is already labeled.

4.2 The Reduce and Verify Phases

The reduction phase is applied to each of the grammars generated by the decompose phase separately. Let G_t be one such grammar. The Reduce function (Box 4) is applied on every pair of trees (T_i, T_j) in G_t where $\text{tagged_skeleton}(T_i) = \text{tagged_skeleton}(T_j)$ or T_i has no adjoining node and $\text{skeleton}(T_i) = \text{skeleton}(T_j)$. Each tree can have a maximum of three distinct non-terminal labels: the root and foot node label, the adjoining node label (if different), and the dummy label 'L' used for all the non-critical nodes. Let X be the label of the root and the foot node of T_i and Y be the label of the root and the foot node T_j , and let Z and W be the labels for the adjoining nodes of T_i and T_j respectively. Without loss of generality, assume that $\text{rank}(X) < \text{rank}(Y)$ and $\text{rank}(Z) < \text{rank}(W)$. Then the Reduce function will replace the occurrences of Y in G_t by X. Z and W are treated similarly unless the W is dummy label 'L'. In this case, it is replaced in the tree under consideration only. In the reduction example illustrated in Figure 5, there is a match between the non-tagged skeletons of T_1 and T_3 . Assuming that $\text{rank}('R') < \text{rank}('Q')$, every label 'Q' will be replaced by 'R'. Also, $\text{rank}('L') = \infty > \text{rank}('P')$. Thus, 'L' is replaced with 'P' in T_3 . Finally, T_3 will be eliminated from the grammar. Let G_{t+1} be the resulting grammar after performing a reduction on G_t . A subset query $\text{SUBSET}(G_{t+1})$ is performed to check if the executed reduction causes the acceptance of any string in \overline{L} . If the query returns false, the reduction is undone.

The success or failure of each reduction trial depends on the order in which the reductions are performed. Thus, the algorithm first arranges the trees, in the grammar to be reduced, in every possible permutation. It then starts comparing and reducing the trees in each of these permutations. Let G_i be a

grammar resulting from the decompose phase, G_{ij} be a certain permutation of the trees in G_i , and G_{ijRed} be the grammar resulting from reducing G_{ij} . Note that $G_{ij} = G_{ik}$ for all possible values of j and k ; however, G_{ijRed} does not necessarily equal G_{ikRed} for all possible values of j and k . After performing all possible reductions for every G_{ij} , $L(G_{ijRed}) \subseteq L(G')$. The verify step uses equivalence queries to check if any of the inferred grammars is equivalent to the unknown grammar G' . The algorithm will output an inferred grammar G such that the query $EQUIV(G)$ returns true.

<p>ExtendedHigh(T)</p> <ul style="list-style-type: none"> - let a be the highest terminal node in T, and b be the lowest node in $Inks(a)$ - let A and B be the parents of a and b respectively - let $C = \text{non-leaf child of } B = \langle p_c, ch_c, \phi, lbl_c, tag_c \rangle$ - let $E = \text{parent}(\text{footnode}(T)) = \langle p_E, ch_E, \phi, lbl_E, tag_E \rangle$ - $TL = \text{Extract}(T, C, E)$ - $TU = T - TL$ - $\text{AdjustBrokenLinks}(TU)$ - let $E_{tu} = \langle p_{E_{tu}}, ch_{E_{tu}}, \phi, lbl_{E_{tu}}, tag_{E_{tu}} \rangle$ where $E_{tu} \in TU$ - set $\text{tag}(E_{tu}) = 1$ - $\text{adjoining_node_label} = \text{AddAuxiliaryTree}(TU)$ - if failure <ul style="list-style-type: none"> - exit; terminate this inference path - set $\text{label}(\text{root}(TL)) = \text{adjoining_node_label}$ - set $\text{label}(\text{footnode}(TL)) = \text{adjoining_node_label}$ - if TL is primitive - $\text{AddAuxiliaryTree}(TL)$ - if failure <ul style="list-style-type: none"> - exit; terminate this inference path - else <ul style="list-style-type: none"> - $\text{Decompose}(TL)$ 	<p>Reduce(T1,T2)</p> <ul style="list-style-type: none"> - let $\text{root_lbl}_i = \text{label}(\text{root}(T_i))$, $i = 1, 2$ - let $\text{adj_lbl}_i = \text{label}(\text{adj_node}(T_i))$, $i = 1, 2$ - if $\text{rank}(\text{root_lbl}_1) < \text{rank}(\text{root_lbl}_2)$ <ul style="list-style-type: none"> - $\text{low_root_lbl} = \text{root_lbl}_1$ - $\text{high_root_lbl} = \text{root_lbl}_2$ - else <ul style="list-style-type: none"> - $\text{low_root_lbl} = \text{root_lbl}_2$ - $\text{high_root_lbl} = \text{root_lbl}_1$ - if $\text{rank}(\text{adj_lbl}_1) < \text{rank}(\text{adj_lbl}_2)$ <ul style="list-style-type: none"> - $\text{low_adj_lbl} = \text{adj_lbl}_1$ - $\text{high_adj_lbl} = \text{adj_lbl}_2$ - else <ul style="list-style-type: none"> - $\text{low_adj_lbl} = \text{adj_lbl}_2$ - $\text{high_adj_lbl} = \text{adj_lbl}_1$ - Replace high_root_lbl and high_adj_lbl with high_root_lbl and high_adj_lbl respectively in all inferred trees and eliminate repetitions - if $\text{SUBSET}(\text{current grammar}) = \text{false}$ <ul style="list-style-type: none"> - undo reduction
---	---

Box 3 ExtendedHigh Decomposition

Box 4 The Reduce function

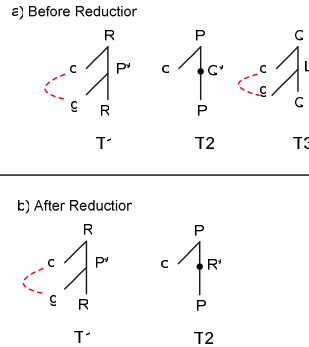


Figure 5 Reduction Example

5. Identification Proof

In this section, we will show that given a structurally complete sample of G' , the decompose and reduce phases will produce at least one grammar G_{red} where $L(G_{red}) = L(G')$.

The decompose phase generates a set of definite grammars. Each grammar G in this set satisfies the condition: $L(G) = S^+$.

Lemma 5.1:

For every skeleton $E \in S^+$ such that $E = A_{init} \text{ adj } A_1 \text{ adj } A_2 \text{ adj } \dots \text{ adj } A_k$, there exists an inference path that generates the set of primitive trees T_{inf} where $\text{TSK}(T_{inf}) = \text{TSK}(\{A_{init}, A_1, A_2, \dots, A_{k-1}\}) \cup \text{SK}(\{A_k\})$.

Proof:

Claim 5.1 can be proved by induction.

Base case:

Assume $T = A_{init} \text{ adj } A_1 \text{ adj } A_2$

The initial tree, A_{init} , will always be inferred correctly; see section 4.1. The other two inferred trees, A_1' and A_2' , will depend on the types of A_1 and A_2 . Notice that A_2' does not have an adjoining node (see the last inferred tree in Figure 4 (e)).

Cases 1-5: A_1 is of type a, and A_2 is of type b, c, d, or e. The inference condition vector = (T,T,T,T,F,F,F,F). The only decomposition option is Outer. A_1' and A_2' will be inferred where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{tagged_skeleton}(A_2') = \text{skeleton}(A_2)$.

Cases 6 – 8 and 10: A_1 is of Type b and A_2 is of type a, b, c, or e. The inference condition vector = (T,F,F,T,F,F,T,T). Possible decomposition options are High and Low. In the path where High is executed, A_1' and A_2' will be inferred where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{tagged_skeleton}(A_2') = \text{skeleton}(A_2)$.

Case 9: A_1 is of Type b and A_2 is of type d. The inference condition vector = (T,F,F,F,F,T,T,T). Possible decomposition options are High and ExtendedHigh. In the path where High is executed, A_1' and A_2' will be inferred where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{tagged_skeleton}(A_2') = \text{skeleton}(A_2)$.

Cases 11-15: A_1 is of Type c and A_2 is of type a, b, c, d, and f. Cases 11-15 are mirror images of cases 6-10. Here, conflicts will occur between High/Low and Low/ExtendedLow. Similar results can be proven.

Cases 16-20: A_1 is of type d and A_2 is of type a, b, c, d, or e. The inference condition vector = (T,F,F,F,F,T,T,T). Possible decomposition options are High and ExtendedHigh. In the path where ExtendedHigh is executed, A_1' and A_2' will be inferred where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{tagged_skeleton}(A_2') = \text{skeleton}(A_2)$.

Cases 21-25: A_1 is of type e and A_2 is of type a, b, c, d, or e. Cases 21-25 are mirror images of cases 16-20. Here, conflicts will occur between Low/ExtendedLow. Similar results can be proven.

Induction Hypothesis: Assume that if $E = A_{init} \text{ adj } A_1 \text{ adj } A_2 \dots \text{ Adj } A_n$, there exists an inference path that will generate the set of primitive trees T_{inf} where $\text{TSK}(T_{inf}) = \text{TSK}(\{A_{init}, A_1, A_2, \dots, A_{n-1}\}) \cup \text{SK}(\{A_n\})$

Induction Step:

Assume that $T = A_{init} \text{ adj } A_1 \text{ adj } A_2 \dots \text{ Adj } A_n \text{ adj } A_{n+1}$

Once again, the initial tree, A_{init} , will always be inferred correctly. The next decomposition step will depend on the types of A_1 and A_2 .

Cases 1-5 : A_1 is of type a and A_2 is of type a, b, c, d, or e. The inference condition vector = (T,T,T,T,F,F,F,F). The only decomposition option is Outer. The output of the decomposition step will be A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{comp}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Case 6: A_1 is of type b and A_2 is of type a. The inference condition vector = (T,F,F,T,F,F,T,F). The only decomposition option is High. The output of the decomposition step will be A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{comp}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Case 7: A_1 is of type b and A_2 is of type b. The inference condition vector = (T,F,F,NA,F,NA,T,NA). Because some of the inference conditions are not determined by the types of A_1 and A_2 in this case, we can not list the possible decomposition options. However, we know that the sufficient conditions for High are satisfied. Thus, one of the inference outputs will be A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{comp}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Case 8: A_1 is of Type b and A_2 is of type c. The inference condition vector = (T,F,F,T,F,F,T,T). Possible decomposition options are High and Low. In the path where High is executed, the output of the decomposition step will be A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{comp}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Case 9: A_1 is of Type b and A_2 is of type d. The inference condition vector = (T,F,F,F,F,T,T,NA). Even with the undetermined values of ic8, the only two possible decomposition options are High and

ExtendedHigh. In the inference path where High is executed, the resulting trees are A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{\text{comp}}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Case 10: A_1 is of Type b and A_2 is of type e. The inference condition vector = (T,F,F,T,F,F,T,T). Possible decomposition options are High and Low. In the path where High is executed, the resulting trees are A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{\text{comp}}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Cases 11-15: A_1 is of Type c and A_2 is of type a, b, c, d, and f. Cases 11-15 are mirror images of cases 6-10. It can be similarly shown that Low will always be one of the decomposition options; thus, similar results can be proven.

Cases 16-20: A_1 is of type d and A_2 is of type a, b, c, d, or e. (T,F,F,F,F,T,T,F/NA/T/NA/T). Irrespective of the value of ic8 which changes with the type of A_2 , in all five cases, the possible decomposition options are High and ExtendedHigh. In the inference path where ExtendedHigh is executed, the resulting trees are A_1' and A_{comp} where $\text{tagged_skeleton}(A_1') = \text{tagged_skeleton}(A_1)$ and $\text{skeleton}(A_{\text{comp}}) = \text{skeleton}(A_2 \text{ adj } A_3 \dots \text{ adj } A_n \text{ adj } A_{n+1})$.

Cases 21-25: A_1 is of type e and A_2 is of type a, b, c, d, or e. Cases 21-25 are mirror images of cases 16-20. It can be similarly shown that ExtendedLow will always be one of the decomposition options; thus, similar results can be proven.

Let us rename the primitive trees A_2, A_3, \dots, A_{n+1} to A_1, A_2, \dots, A_n . Considering the fact that the decomposition process depends only on the skeleton of the tree (no tags or non-terminal labels required), and using the induction step, we conclude our claim. \square

Note: We will refer to the inference path that follows the decomposition options resulting in a solution that satisfies Lemma 5.1 as the correct inference path.

Lemma 5.2

Let S^+ be a structurally complete sample of the language $L(G')$. Let \mathcal{G} be the set of all grammars resulting from the decompose phase of the algorithm given S^+ . There exists at least one inferred solution $G = T_{\text{inf}} \cup ET_{\text{inf}} \in \mathcal{G}$ such that $\text{TSK}(T_{\text{inf}}) = \text{TSK}(G')$ and $\text{TSK}(ET_{\text{inf}}) \subseteq \text{SK}(G')$.

Proof:

Let the set of trees generated by the correct inference path upon receiving positive example $E_i = \text{skeleton}(A_{\text{init}} \text{ adj } A_1 \text{ adj } A_2 \text{ adj } \dots \text{ adj } A_k)$ be $T_{\text{inf}}(i) \cup ET_{\text{inf}}(i)$ where all the trees in $T_{\text{inf}}(i)$ have an adjoining node and $ET_{\text{inf}}(i)$ has one tree with no adjoining node. The tree in $ET_{\text{inf}}(i)$ matches in skeleton the tree A_k . We will refer to A_k as an end tree. Then $T_{\text{inf}} = \bigcup_{E_i \in S^+} T_{\text{inf}}(i)$ and $ET_{\text{inf}} = \bigcup_{E_i \in S^+} ET_{\text{inf}}(i)$. Using Lemma 6.1, we

conclude that $\text{TSK}(T_{\text{inf}}) \subseteq \text{TSK}(G')$ (1) and $\text{TSK}(ET_{\text{inf}}) \subseteq \text{SK}(G')$ (2). Since S^+ is a structurally complete sample, each tree will appear at least in one example as a non end tree. Thus, $\text{TSK}(G') \subseteq \text{TSK}(T_{\text{inf}})$ (3). From (1), (2), and (3), we conclude Lemma 5.2 \square

Observation 5.1: The reduction phase is a generalization step that introduces recursion into the definite grammars generated from the previous phase. This generalization broadens the language accepted by each grammar. Because we use subset queries to confirm reductions, we know that after the reduction phase each grammar G in the solution set satisfies the condition $S^+ \subseteq L(G) \subseteq L(G')$.

Lemma 5.3

Let S^+ be a structurally complete sample of the language $L(G')$. Let \mathcal{G}_{Red} be the set of all grammars resulting from the decompose and reduce phases of the algorithm given S^+ . There exists at least one inferred solution $G_{\text{Red}} \in \mathcal{G}_{\text{Red}}$ such that $L(G_{\text{Red}}) = L(G')$.

Proof:

From Lemma 5.2 we conclude that given a structurally complete sample, there exists at least one inferred solution $G \in \mathcal{G}$ such that $G = T_{\text{inf}} \cup ET_{\text{inf}}$ where $\text{TSK}(T_{\text{inf}}) = \text{TSK}(G')$ and $\text{TSK}(ET_{\text{inf}}) \subseteq \text{SK}(G')$. Because S^+ is a structurally complete sample and every tree in G' has exactly one adjoining node, every tree in G' resulted in inferring one or more trees in $T_{\text{inf}} \dots$ (1). Because of the way G is constructed, as a definite grammar for a structurally complete sample and from (1), G can be reduced to G_{red} where $G' \subseteq G_{\text{red}}$ if the tree reduction happens in a certain “correct” order. Because we do not know the “correct” reduction order that will ensure such result, the algorithm performs the reductions in each possible ordering of the trees. Thus, the reduction of G will result in at least one grammar G_{red} where $G' \subseteq G_{\text{red}}$. This means that $L(G') \subseteq L(G_{\text{red}}) \dots$ (2). From **Fact 5.1** and (2) we conclude that $L(G_{\text{red}}) = L(G')$ \square .

6. Time Complexity

Decompose Phase: Since each primitive tree has a yield of at least one character, the size of each generated grammar is $O(n)$ trees. The time required for each decomposition operation to evaluate the inference conditions, perform the extract, subtract, and adjust link operations, and check if the resulting primitive tree violates the model conditions is a linear function in the maximum size of a grammar tree yield which is $O(n)$. Thus, the total time required for the decomposition phase is $O(n^2 M)$ where M is the number of generated grammars.

Reduce Phase: Reduction is performed for each of the M grammars. It is performed on each possible permutation or ordering of trees in each of the grammars. The size of each grammar is $O(n)$. Thus, there are $O(n!)$ permutations of the trees in each grammar. We will refer to the process of reducing the trees in each of these permutations as a reduction process. In each reduction process, $O(n^2)$ comparisons are required to determine the skeleton matches. The time required for each comparison is a linear function of the maximum size of a grammar tree yield which is $O(n)$. Each reduction process will perform up to $O(n)$ reductions, each of which will require updating labels of up to $O(n)$ trees. Tree updating requires constant time. After each reduction one SUBSET query is performed. Thus, the total time for one reduction process is $O(n^3 + n^2)$, and the total time for the whole phase is $O(n^3 M n!)$. This is a conservative estimate since the equivalence query might come up with the YES answer early on. If many of these grammars are the correct answers, then a random grammar has a good probability of yielding the answer YES on an equivalence query. Also, the number of permutations of trees in a given grammar could be much less than $n!$.

Verify Phase: The verify phase simply executes an EQUIVALENCE query for each of the reduced grammars. The total time for this phase is $O(M n!)$. Thus the total time for the all three phases is $O(n^3 M n!)$. Clearly, this run time is very generous. We would like to emphasize that the main focus of this paper is to introduce the new grammatical formalism suitable for RNA structures and demonstrate the feasibility of inferring this grammar from samples.

Analyzing M: At any point in time during the decompose phase, there could be at most two decomposition options. Thus, very conservatively we can say that the decompose phase will result in $O(2^n)$ solutions in the worst case. However, with further analysis, it might be possible to deduce a tighter bound for M .

7. Conclusion

In this paper, we have defined a subclass of TAGs, Linked Single Adjoining (LSA)-TAG², which can be learned from structural positive examples and queries. We presented an inference algorithm for LSA-TAG². The algorithm starts by decomposing the skeleton in the sample to a set of elementary trees generating a definite grammar that can only accept the sample. At certain points in the decomposition

process, more than one decomposition option may seem plausible, so more than one solution is maintained. After decomposition is done, a sequence of reductions is performed and subset queries are used to validate the reductions. Finally, equivalence queries are used to choose one of the resulting grammars after reduction.

Recently, there has been a special focus on the use of grammatical inference in bioinformatics [5][13][19]. In this direction and driven by our motivation to use this algorithm to build grammars for RNA sequences, the following are three research problems that we are planning to address next.

- Developing a method to construct the linked skeleton for RNA sequences assuming a generic grammar similar to TAG^2_{RNA} . This will enable us to use the algorithm for data extracted from RNA databases.
- Experimentally studying the number of solutions generated by the decompose phase, and trying to enhance the worst-case complexity, possibly by using heuristic methods.
- Carefully studying the reduction process to reach an understanding of what makes a special order of reductions “correct” and consequently reducing the complexity of the reduction phase.
- Developing a heuristic version of the algorithm that does the inference using only positive data. We do not expect this version to be able to do identification in the limit. We plan to study the quality of the output it generates experimentally.

On the theoretical level, we are currently working on studying the formal linguistic properties of LSA-TAG².

We would like to conclude with a mention of two open problems.

- Developing a tighter bound for the number of grammars generated by the decompose phase.
- Extending the model and the inference algorithm to deal with TYPE5 trees of TAG^2_{RNA} .

References

- [1] D. Angluin, “Queries and Concept Learning,” *Machine Learning* 2: 319-342, 1988.
- [2] D. Angluin, “Learning Regular Sets from Queries and Counter Examples,” *Information and Computation*, 75:87-106, 1987.
- [3] R. Alquézar, A. Sanfeliu, “Recognition and Learning of a Class of Context-Sensitive Languages Described by Augmented Regular Expressions”, *Pattern Recognition*, 30 (1):163-182, 1997.
- [4] L.Becerra-Bonache and T. Yokomori, “Learning Mild Context-Sensitiveness: Toward Understanding Children's Language Learning,” in: G. Paliouras and Y. Sakakibara (Eds.), *Grammatical Inference: Algorithms and Applications, Proceedings of ICGI '04, Lectures Notes in Computer Science*,3264:53-64, 2004.
- [5] A. Brazma, I. Jonassen, J. Vilo and E. Ukkonen, “Pattern Discovery in Biosequences,” *Proceedings of ICGI '98, Lecture Notes in Computer Science*,1433: 255-270, 1998
- [6] K. Fu and T. Booth, “Grammatical Inference : Introduction and Survey – Part I,” *IEEE Transactions on Systems, Man, and Cybernetics*, July 1975.
- [7] E. Gold, “Language Identification in the Limit,” *Information and Control*, 10(5): 447-474, 1967.
- [8] K. Joshi. “Factoring Recursion and Dependencies: an Aspect of TAG and a Comparison of Some Formal Properties of TAGs, GPSGs, PLGs and LFGs”. In: *21st Annual Meeting of the Association for Computational Linguistics*, pp. 7--15, Cambridge, MA, 1983.
- [9] A. K. Joshi, L. Levy, & M. Takahashi, “Tree Adjunct Grammars,” *Journal of Computer and System Sciences* 10: 136–163, 1975.
- [10] A. K. Joshi and Y. Schabes. “Tree-Adjoining Grammars,” In Grzegorz Rosenberg and Arto Salomaa, (Eds.), *Handbook of Formal Languages and Automata* 3:69–124. Springer-Verlag, Heidelberg, 1997.
- [11] Ker-I Ko, A. Marron. “Identification of Pattern Languages from examples and queries”. *Information and Control*. 74: 91-112 (1987).
- [12] S. Kobayashi and T. Yokomori, “Modeling RNA Secondary Structures Using Tree Grammars,” *Proceeding of Genome Informatics Workshop V*, Universal Academy Press, 29-38, 1994.
- [13] J.A.Laxminarayana, G.Nagaraja, P.V. Balaji, “Identification of Pseudoknots in RNA Secondary Structures : A Grammatical Inference Approach,” *Proceedings of Fifth International Conference on Advances in Pattern Recognition*, 2003.

- [14] J. Oncina and P. Garcia, "Inferring Regular Languages in Polynomial Update Time" in: N. Perez de la Blanca, et al. (Eds.), *Pattern Recognition and Image Analysis, Series in Machine Perception and Artificial Intelligence*, 1: 49-61, 1992.
- [15] Y. Sakakibara, *Algorithmic Learning of Formal Languages and Decision Trees*, Doctoral Dissertation, Department of Information Technology, Tokyo Institute of Technology, 1991.
- [16] Y. Sakakibara, "Efficient Learning of Context-Free Grammars from Positive Structural Examples," *Information and Computation*. 97: 23-60, 1992.
- [17] Y. Sakakibara, "Recent Advances in Grammatical Inference" *Theoretical Computer Science* 185: 15-45, 1997.
- [18] Y. Sakakibara, and H. Muramatsu, "Learning context-free grammars from partially structured examples," in: A. de Oliveira (Ed.), *Grammatical Inference: Algorithms and Applications, Proceedings of ICGI '00, Lectures Notes in Artificial Intelligence*, 1891: 229-240, 2000.
- [19] Y. Sakakibara, "Grammatical Inference in Bioinformatics", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27: 1051-1062, 2005.
- [20] Y. Uemura, A. Hasegawa, S. Kobayashi, and T. Yokomori, "Tree Adjoining Grammars for RNA Structure Prediction", *Theoretical Computer Science*, 210(2):277-303, 1999.