Derived Trees Evolution for Tree Adjoining Grammars Parsing

ADRIAN HORIA DEDIU
Faculty of Engineering in Foreign Languages
University “Politehnica” of Bucharest, Romania
Splaiul Independentei, 313, Sect. 6, Cod 77206, Bucharest, Romania
E-mail: hd@ziplip.com

CARLOS MARTÍN-VIDE¹
Research Group on Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
E-mail: cmv@astor.urv.es

Abstract
Starting from the model proposed by means of Grammatical Evolution, we extended the applicability of the parallel and cooperative searching processes of Evolutionary Algorithms to a new topic, Tree Adjoining Grammars parsing. We evolved derived trees using a string-tree-representation. We also could use a linear matching function to compare the yield of a derived tree with a given input and the running tests present several encouraging results. Due to the higher degree of intrinsic parallelism Evolutionary Algorithms represent a new alternative to the highly complex classical methods used in Formal Computational Models.

1 Introduction
Context-free grammars (CFGs) are a well known class of grammars extensively used for programming languages and they can also describe almost all structures of natural languages. Yet for several exceptions like multiple agreement languages \( \{a_1 a_2 \ldots a_k | n \geq 1, k \geq 3\} \), copy languages \( \{ww | w \in \{a, b\}^*\} \) and cross agreement \( \{a^n b^m, c^n, d^m | n, m \geq 1\} \), the context-free grammars are not the most appropriate investigation instrument for natural languages analysis. Tree-Adjoining Grammars, or TAGs for short, were introduced by Joshi, Levy and Takahashi in 1975 to model some linguistic aspects. Tree-Adjoining Grammars (TAGs) is an important class of

¹Work done in the framework of the Contract No ICA1-CT-2000-70024 of the European Community
grammars, originally motivated by some linguistic considerations, which later have yielded some important mathematical and computational results, which in turn had linguistic implications. A.K. Joshi and Y. Shabes published an overview of TAGs in [8].

Parsing algorithms play an important role in the implementation of compilers, interpreters for programming languages or natural language processing. Parsing usually refers to the construction of a derivation tree. It is also possible to decide if a string belongs or not to a given language (the membership problem) without constructing the derivation tree. Numerous parsing algorithms have been developed over the years. Two of the most well known parsing algorithms for CFGs are CYK recognizer [5] and Earley parser [4].

A recognizer is an algorithm that takes as input a string and either accepts it or rejects it, depending on whether the input string belongs or not to the language of a grammar. In Figure 1 we can see a recognizer data flow.

A parser is a recognizer which also outputs the derivation trees if the string is accepted by the grammar.

Various parsing algorithms for TAGs were proposed and probably Joshi’s algorithms represent the reference points in TAG parsing, [7] whose worst time complexity is \( O(n^9) \) and [8] with the complexity \( O(n^6) \).

We can see that due to the high complexity of the classical TAG parsing algorithms, long sentences could represent a very difficult task to process, as an example if we consider more than 15 words then the worst case complexity is quite a large number. Evolutionary Algorithms for TAGs parsing are able to process such long sentences. Our variant of fitness function evaluation having a linear complexity can increase the limit of the parsed words per sentence.

2 Definition, Components and Composition Operations in Tree Adjoining Grammars

Definition 2.1 A tree- adjoining grammar is a 5-tuple \((T, N, I, A, S)\) where:

1. \( T \) is the alphabet used to build up a language (finite set of terminal symbols)

2. \( N \) is the set of non-terminal symbols (variables)
3. $S$ is the start non-terminal symbol of the grammar

4. $I$ is a finite set of finite trees, which are called initial trees, and they have the following features:

- the interior nodes are non-terminal symbols
- the nodes on the frontier of the initial trees are terminal or non-terminal symbols; the non-terminal symbols on the frontier which can be substituted are marked with a down arrow (↓)

5. $A$ is a finite set of finite trees, which are called auxiliary trees, and they have the following features:

- interior nodes are non-terminal symbols
- the nodes on the frontier are terminal or non-terminal symbols
- The nodes on the frontier are marked for substitution (↓) except for the foot node (annotated with an asterisk *). The label of the foot node must be identical with the label of the root node.

A TAG where at least one terminal symbol (anchor) appears at the frontier of every initial or auxiliary tree is called a lexicalized TAG.

The trees in $I \cup A$ are called elementary trees.

The trees with root labeled by the nonterminal $A$ are called $A$-type trees.

We may associate with every node in a tree a node address (Gorn-position) in an inductive way. The root node has the empty address. For a child node we take the parent address and we add a dot then the children number counted from left. In the set of elementary trees we can form a global address using tuples formed by $(\text{treeName}, \text{nodeAddress})$.

There are also alternative ways to assign addresses to nodes of trees. For instance, we can assign sequential numbers to nodes traversing recursively the tree in a root-left-right or left-root-right manner. We even can count sequentially the nodes in all the trees.

TAGs operate with two composition operations, adjoining and substitution.

**Definition 2.2** Substitution is an operation that takes a nonterminal node $A$ marked for substitution by a down arrow (↓) which is located on the frontier of a tree and replaces it with a tree from the initial trees whose root has the same label as the node $A$.

**Definition 2.3** Adjoining builds a new tree from an auxiliary tree $\beta$ and another tree $\alpha$, which can be initial, auxiliary or a derived tree. Let us consider that the root node of the auxiliary tree $\beta$ is labeled by $X$ (also the foot node, by definition). If an internal node of the tree $\alpha$ is labeled by $X$ then the adjoining operation will construct a tree as follows:

- the sub-tree of $\alpha$ dominated by $X$, call it $t$, is removed from the tree $\alpha$, leaving a copy of $X$ behind
the auxiliary tree $\beta$ is attached to the copy of the node labeled by $X$ in the excised tree $\alpha$.

- the sub-tree $t$ is attached to the foot node of $\beta$ and the root of $t$ is identified with the foot node of $\beta$ (they have the same label $X$).

By definition, any adjoining on a node marked for substitution is forbidden.

For linguistic reasons, we need a more precise way to specify which auxiliary tree can be adjoined at a given node, therefore several constraints on adjoining were introduced.

**Definition 2.4** In a TAG $G = (T, N, I, A, S)$, for each node of an elementary tree on which adjoining operation is allowed we can specify one of the following three constraints on adjunction:

- **Selective Adjunction** written as $\text{SA}(AT)$ specifies a set of trees $AT \subset A$, the set of auxiliary trees that can be adjoined in a given node.
- **Null Adjunction** written as $\text{NA}$ forbids any adjunction on a given node. We have $\text{NA}=\text{SA}(\emptyset)$
- **Obligatory adjunction** written $\text{OA}(AT)$ specifies a set of trees $AT \subset A$, from which one of the trees is mandatory to be inserted on a given node.

If there are no substitution nodes and no constraints, then we have a pure TAG.

After applying the adjoining or the substitution operations we obtain the derived trees. The derived trees do not have the information regarding how they were build, therefore we should build a special structure able to specify how a derived tree was constructed.

**Definition 2.5** A TAG derivation tree is a tree used to show how a derived tree was constructed and it has:

- The root labeled by an $S$-type initial tree
- all other nodes are labeled with trees and parents’ nodes addresses where the composition operation (substitution / adjoining) has been performed.
- the arcs in the derivation tree connect a node labeled with (tree, parent’s node address) with the parent tree. The substitution arcs are dashed and the adjoining arcs are continuous.

An initial tree is completed if there is no substitution node on the frontier of it and for all the obligatory adjunction constrains are satisfied.

The tree set, $T_G$ of a TAG is defined as the set of the completed initial trees derived from some $S$-rooted initial tree.

The string language, $L(G)$, of a TAG is the set of yields of all the trees in the tree set $T_G$: $L(G) = \{w | w = \text{yield}(t), t \in T_G\}$
3 Lexicalized Grammars

Lexicalized Grammars presented in [8] by Joshi and Shabes have both linguistic and formal importance.

Definition 3.1 A grammar is lexicalized if it consist of:

- a finite set of structures, each one associated with a lexical item called the anchor of the corresponding structure
- one or more operations for composing the structures

The anchor must not be the empty string.

Proposition 3.1 Lexicalized grammars are finitely ambiguous.

We observe that a finite sentence has a finite number of lexical items and hence a finite number of structures attached to the lexical items. The finite number of structures may be combined in finitely many ways to produce compound structures. Therefore we have a finite number of derivations that produce the initial sentence. For this reason we have:

Proposition 3.2 It is decidable whether or not a string is accepted by a lexicalized grammar.

Definition 3.2 We say that a formalism $F$ can be lexicalized by another formalism $F'$, if for any finitely ambiguous grammar $G$ in $F$ there is a grammar $G'$ in $F'$ s.t. $G$ and $G'$ generate the same tree set and $G'$ is lexicalized.

In general CFGs are not in lexicalized form because not all the rules contain in the right hand side a lexical item and sometimes CFGs could be infinitely ambiguous, containing recursive derivation chains such as $X \Rightarrow^* X$. Lexicalization of finitely ambiguous CFGs achieved by transforming them into the Greibach Normal Form [5], can be regarded as weak lexicalization since we do not preserve also the structure of the derivation trees. The previous definition of lexicalization may be regarded as strong lexicalization.

Definition 3.3 A tree-substitution grammar (TSG) is a TAG without auxiliary trees and without the adjoin operation.

We will use the same terminology for derived trees and derivation trees. A tree is completed if on its frontier there are only terminals. We will denote with $t_X$ an $X$-type initial tree and with $Fr(t_X)$ the frontier of the tree $t$.

The set of languages generated by TSGs is the same as the set of languages generated by CFGs. It is easy to see that for a TSG there is an equivalent CFG that generates the same language. For any initial tree $t_X$ in the TSG we write a CFG production of the form $X \rightarrow Fr(t_X)$. For the other inclusion we take a CFG grammar and for any production, we construct an initial tree with the root the nonterminal
in the left hand side of the production, having the children the terminals and non-
terminals from the right hand side of the production. The non-terminals on the
frontier of the initial tree are marked for substitution.

The following propositions are well known from literature [8].

**Proposition 3.3** Finitely ambiguous CFGs cannot be lexicalized by a TSG.

**Proposition 3.4** A finitely ambiguous CFGs which does not generate the empty
string may be lexicalized by a TAG without substitution nodes.

We observe that adjunction is sufficient to lexicalize CFGs, but using substitution
as an additional operation we may obtain more compact TAGs.

## 4 Classical Tree Adjoining Grammars Recognizer

We will present a version of an Earley algorithm for TAGs described by Joshi and
Schabes in [8]. The original algorithm uses a chart of items. Every item contains a
dotted tree and the dot may be in one of the following positions with respect to a
node in the tree: left-and-above, left-and-bellow, right-and-bellow, right-and-above.
Because of this representation, in the chart there might appear several equivalent
items with different representation as we can see in Figure 2, for every dotted tree
there could be two equivalent items in the chart having right-and-above and right-
and-bellow equivalent dot positions on the left picture and right-and-above and left-
and-above equivalent dot positions on the right picture.

![Figure 2: Equivalent dot positions](image)

We will introduce a shorthand notation for trees’ structures, namely string rep-
resentation for trees, in order to simplify trees’ description. Let us consider $n$, an
internal node inside a tree, as an open bracket having as subscript the label of the
node $n$. Then we will write all the children of the node $n$ followed by a closed bracket.
If we can distinguish rapidly the nonterminals from the terminals, using respectively
uppercase letters and lowercase letters from the alphabet, then we can simplify even
more the notation, by renouncing to the open brackets.

Using the string-representation for trees we reduced the state description for an
item because instead of a dotted tree and a position of the dot we have only a dotted
string-tree. Supplementary the problem of equivalent dotted items disappears, as
we can see the trees in the Figure 2 have the following representation: $Se.]$ and $S^{NA}a.SbS^{NA}c[d]$.

4.1 Informal Description of a TAG (LTAG) Recognizer

Trying to reduce the number of trees in grammars, practical considerations imposed the usage of lexicalized TAGs (LTAGs) instead of TAGs. For the algorithms described in this paper the Tree Adjoining Grammars should not be lexicalized but the implementations could benefit from the fact that the grammars are lexicalized.

We define an item $s$ as a 7-tuple,

$$s = [\text{treeName}, \text{dottedStringTree}, i, j, k, l, \text{sat}]$$

where:

- $\text{treeName}$ is the name of an elementary tree
- $\text{dottedStringTree}$ is the string-representation of the tree $\text{treeName}$
- $i, j, k, l$ are indices of positions in the input string ranging between -1 and $n$, $n$ being the length of the input string. Only the indices $j$ and $k$ may have the -1 value and this means that they are not bounded. The index $l$ is used to point the last analyzed character in the input string.
- $\text{sat}$? is used to disallow more than one adjunction in the same node. $\text{sat}$? takes values ranging from $\{\text{nil, true}\}$, where $\text{nil}$ means no adjunction yet performed on the dotted node and $\text{true}$ means an adjunction was performed on the dotted node.

According to the recognizer algorithm’s necessities, the dot may be positioned in front of the foot symbol (*) or after.

Initially, for all $S$-type initial trees $\alpha_i$, the chart $C$ contains all items of the form:

$$[\alpha_i, \text{"."+}\alpha_i\text{StringTree}, 0, -1, -1, 0, \text{nil}]$$

Depending on the existing items in the chart $C$, new items are added in the chart. According to the items in the chart four basic operations add new items: PREDICT, SCAN, COMPLETE and ADJOIN that work basically as described in [8]. The algorithm stops in two cases:

- if no items can be added to the chart then the input string was not recognized
- if an item of the form:

$$[\alpha, \alpha\text{StringTree}+\text{"."}, 0, -1, -1, n, \text{nil}]$$

where $\alpha$ is an $S$-type initial tree is to be added then the input string was recognized. It is normal for indices $j$ and $k$ to have the value -1 since in the item we refer to an initial tree without a footer node.
Table 1: New items added in the chart

<table>
<thead>
<tr>
<th>Functional description</th>
<th>After dot</th>
<th>Current item</th>
<th>Supplementary conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scan 1</strong></td>
<td>$a$</td>
<td>$(t_1, s_1+a^<em>a^</em>+s_2, i, j, k, l, \text{nil})$</td>
<td>inputString[\text{i+1}]=a</td>
</tr>
<tr>
<td><strong>Added item</strong></td>
<td></td>
<td>$(t_1, s_1+a^<em>a^</em>+s_2, i, j, k, l+1, \text{nil})$</td>
<td></td>
</tr>
<tr>
<td><strong>Scan 2</strong></td>
<td>$\lambda$</td>
<td>$(t_1, s_1+\lambda^*+\lambda+s_2, i, j, k, l, \text{nil})$</td>
<td></td>
</tr>
<tr>
<td><strong>Added item</strong></td>
<td></td>
<td>$(t_1, s_1+\lambda^*+\lambda+s_2, i, j, k, l, \text{nil})$</td>
<td></td>
</tr>
<tr>
<td><strong>Predict 1</strong></td>
<td>$V$</td>
<td>$(t_1, s_1+\cdot+V+s_2, i, j, k, l, \text{nil})$</td>
<td>for each $at \in Adj(t_1, \text{pos}(V))$</td>
</tr>
<tr>
<td><strong>Added items</strong></td>
<td></td>
<td>$(at, \cdot+\text{str}(at), i, j, k, l, \text{nil})$</td>
<td></td>
</tr>
<tr>
<td><strong>Predict 2</strong></td>
<td>$V$</td>
<td>$(t_1, s_1+\cdot+V+s_2, i, j, k, l, \text{nil})$</td>
<td>$OA(V) = false$</td>
</tr>
<tr>
<td><strong>Added item</strong></td>
<td></td>
<td>$t_1, s_1+V^*+\cdot+s_2, i, j, k, l, \text{nil}$</td>
<td></td>
</tr>
<tr>
<td><strong>Predict 3</strong></td>
<td>*</td>
<td>$(t_1, s_1+V^*+\cdot+s_2, i, j, k, l, \text{nil})$</td>
<td>for each $t$ in trees, $t \in C$</td>
</tr>
<tr>
<td><strong>Complete 1</strong></td>
<td>]</td>
<td>$(t_1, s_1+\cdot+\cdot+s_2, i, j, k, l, \text{nil})$</td>
<td>for each $it \in C$</td>
</tr>
<tr>
<td><strong>Added items</strong></td>
<td></td>
<td>$(t_2, s_3+V^*+\cdot+s_4, i, j, k, l, \text{nil})$</td>
<td>$\text{(s3+\cdot+s4, i, j, k, l, nil)}$</td>
</tr>
<tr>
<td><strong>Complete 2</strong></td>
<td>]</td>
<td>$(t_1, s_1+\cdot+\cdot+s_2, i, j, k, l, \text{nil})$</td>
<td>for each $it \in C$</td>
</tr>
<tr>
<td><strong>Added items</strong></td>
<td></td>
<td>$(t_1, s_1+\cdot+\cdot+s_2, h, j', k', i, sat?)$</td>
<td>$\text{(s3+\cdot+s4, i, j, k, l, nil)}$</td>
</tr>
<tr>
<td><strong>Adjoin</strong></td>
<td></td>
<td>$(t_1, V+s1+\cdot+\cdot+s2+\cdot+s_2+\cdot+s_2, i, j, k, l, \text{nil})$</td>
<td>for each $it \in C$</td>
</tr>
<tr>
<td><strong>Added items</strong></td>
<td></td>
<td>$(t_2, s3+\cdot+s4, i, j, k, l, true)$</td>
<td>$\text{(t2, s3+\cdot+s4, i, j, k, l, true)}$</td>
</tr>
</tbody>
</table>
4.2 Formal Description of a TAG (LTAG) Recognizer

We will use for the nodes’ addresses the index in the string-tree representation and the function \( \text{pos} \) to return the node number in the string-tree representation.

We will use the following notations: \( \text{Var} \) is the set of variables; \( \text{Term} \) is the set of terminals; \( \text{StartingSymbol} \) is the value of the starting symbol; \( \text{trees} \) is the set of all elementary trees; for a given tree \( t \) we will write \( \text{str}(t) \) the string-tree representation of the tree \( t^2 \); \( \text{NullAdjoining} \) is the set of pairs \((\text{treeName}, \text{nodeAddress})\) for which the \( \text{NullAdjoining} \) attribute is set; \( \text{Adj}(\text{treeName, nodeAddress}) \) is the set of auxiliary trees that can be adjoined in the \( \text{treeName} \) at the address \( \text{nodeNumber} \); \( \text{inputString} \) is the value of the string to be recognized and we have the \( \text{inputString} = a_1 \ldots a_n \); \( \text{root} \) is a function that has one argument as a string-tree \( t \) and returns the nonterminal character that is the root of the tree \( t \); \( \text{max}(i, j) \) is the maximum value between \( i \) and \( j \); \( \text{openBracket} \) is a function that returns the non-terminal corresponding to a certain closed bracket in the string-tree representation that may be described as follows:

\[
\text{counter} = 1
\]

Starting from the dot position,
go left in the dotted string-tree,
increment the counter for any “]”,
decrement the counter for any \( V \) in \( \text{Var} \),
if counter=0 then return \( V \)

Internally the algorithm builds the \( \text{initialTrees} \) and \( \text{auxiliaryTrees} \) sets based on the \( \text{trees} \) set. Supplementary, we will use the \( \text{item} \) definition as described in the informal description. The recognition algorithm works with a set of items collected in a chart \( C \). Also we will define a procedure \( \text{chartAdd} \) that adds an item to the chart \( C \) only if the item is not already in the chart.

The pseudo-code for the TAG recognizer algorithm may be described as follows:

```plaintext
function recognize
    \( C = \emptyset \)
    for each \( t \) in \( \text{initialTrees} \)
        if \( \text{root}(t) = \text{StartingSymbol} \)
            \( \text{chartAdd}([t, "", "stringDescription(t), 0, -1, -1, 0, nil]) \)
        end if
    next \( t \)

apply for each \( i \) in \( C \)
    if (\( \text{addNewItems}(i) = \text{"stop"} \)) then
        return("recognized")
    end if
until no more items are added in the chart \( C \)
return("not recognized")
end function
```

\(^2\)We can distinguish between initial trees and auxiliary trees, because auxiliary trees will contain the footer symbol "*"
The function addNewItems(currentItem) returns "stop" if a new item of the form:

\[[\alpha, \alphaStringTree^+.\cdot, 0, -1, -1, n, nil]\]

where \(\alpha\) is an \(S\)-type initial tree is to be added in the chart \(C\).

The function addNewItems considers the current item in the chart, and according to the character that follows after dot in the dotted string tree description and a supplementary condition, adds new items in the chart.

To reduce the space used by variable description we will use \(NA\) and \(OA\) for NullAdjoining, ObligatoryAdjoining sets, respectively. We will use also \(a\) for a terminal \(a \in \text{Term}\), \(V\) for a variable \(V \in \text{Var}\), and \(\leftarrow\) for the function openBracket.

### 4.3 Running Example for the classical TAG (LTAG) Recognizer

We will illustrate the recognizer’s work using an example. To describe the corresponding TAGs, we will use the formalism that we already introduced.

**Example 4.1**

The Table 2 describes the TAG \(G_2 = (N = \{S\}, T = \{a, b, c, d, e\}, I = \{\alpha : Se\}, A = \{\beta : SNAaSbS^N\alpha^*c][d]\}, S)\).

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>set</td>
<td>{S}</td>
</tr>
<tr>
<td>Term</td>
<td>set</td>
<td>{a,b,c,d,e}</td>
</tr>
<tr>
<td>StartingSymbol</td>
<td>value</td>
<td>S</td>
</tr>
<tr>
<td>trees</td>
<td>(\alpha)</td>
<td>Se]</td>
</tr>
<tr>
<td>trees</td>
<td>(\beta)</td>
<td>SNaSbS*]c][d]</td>
</tr>
<tr>
<td>NullAdjoining</td>
<td>(\beta)</td>
<td>(1,5),</td>
</tr>
<tr>
<td>ObligatoryAdjoining</td>
<td>set</td>
<td>{}</td>
</tr>
<tr>
<td>inputString</td>
<td>value</td>
<td>aabbeccdd</td>
</tr>
</tbody>
</table>

The recognizer fills in the chart as described in the Table 3.

### 4.4 Complexity of the TAG (LTAG) Recognizer Algorithm

The presented algorithm has the worst case time complexity \(O(|A| \cdot |A \cup I| \cdot M \cdot n^6)\) where \(|A|\) is the number of auxiliary trees, \(|A \cup I|\) is the number of elementary trees, \(M\) is the maximum number of nodes in an elementary tree and \(n\) is the length of the input string. In LTAGs, we can select from the whole grammar only those trees that have lexical anchors in the parsing sentence hence we can reduce dramatically the number of elementary trees used in the parsing process. The worst case complexity is
Table 3: The chart after the recognition of the TAG $G_2$ and the input string $aabbeccdd$

<table>
<thead>
<tr>
<th>Item</th>
<th>No.</th>
<th>Value</th>
<th>Apply</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1(1)</td>
<td>2</td>
<td>1, 4</td>
<td>P1+2; P3+2</td>
</tr>
<tr>
<td>P2(1)</td>
<td>3</td>
<td>0, 1, 3</td>
<td>S1+2</td>
</tr>
<tr>
<td>S1(4)</td>
<td>5</td>
<td>1, 1</td>
<td>P1+6; P2+7</td>
</tr>
<tr>
<td>P1(5)</td>
<td>6</td>
<td>1, 1</td>
<td>P2+3</td>
</tr>
<tr>
<td>P2(5)</td>
<td>7</td>
<td>1, 1</td>
<td>S1+2</td>
</tr>
<tr>
<td>P2(6)</td>
<td>8</td>
<td>0, 1, 3</td>
<td>S1+2</td>
</tr>
<tr>
<td>S1(8)</td>
<td>9</td>
<td>1, 2</td>
<td>P1+10; P2+11</td>
</tr>
<tr>
<td>P1(9)</td>
<td>11</td>
<td>0, 1, 3</td>
<td>S1+2</td>
</tr>
<tr>
<td>P1(10)</td>
<td>12</td>
<td>0, 1, 3</td>
<td>S1+2</td>
</tr>
<tr>
<td>P2(13)</td>
<td>14</td>
<td>0, 1, 3</td>
<td>S1+2</td>
</tr>
<tr>
<td>P2(17)</td>
<td>18</td>
<td>0, 1, 3</td>
<td>S1+2</td>
</tr>
<tr>
<td>C1(21)</td>
<td>21</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(22)</td>
<td>23</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C1(24)</td>
<td>25</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(25)</td>
<td>26</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>S1(26)</td>
<td>27</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(27)</td>
<td>28</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>S1(28)</td>
<td>29</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(29)</td>
<td>30</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>A(30)</td>
<td>31</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(31)</td>
<td>32</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>S1(32)</td>
<td>33</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(33)</td>
<td>34</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>A(34)</td>
<td>35</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
<tr>
<td>C2(35)</td>
<td>36</td>
<td>1, 4</td>
<td>S1+2</td>
</tr>
</tbody>
</table>

The positions in the input string are: $aabbeccdd$
reached during the ADJOIN operation. As we can observe in the Table 1 during the
adjoin operation we have to combine two items \((t_1, \ldots, i, j, k, l, \text{nil})\) and \((t_2, \ldots, j, p, q, k, \text{nil})\), therefore we have at most \(n^6\) instances of indices \((i, j, k, l, p, q)\) and we
can call the adjoin operation \(|A| \cdot |A \cup I| \cdot M\) times.

5 Evolutionary Algorithms for
Tree Adjoining Grammars Parsing

We can find an introduction in Evolutionary Algorithms and their applications in
[2] and [3] and a very good theory overview in [1]. Grammatical Evolution (GE)
proposed by a group from the University of Limerick [6], combines aspects of Context
Free Grammars with the searching capabilities of Evolutionary Algorithms in order
to evolve high-level languages. GE orders the productions for every non-terminal in a
CFG and then uses the gene values in order to decide which production to use when
it is necessary to expand a given non-terminal.

There are two main problems solved by GE. First, we might have an invalid gene
value when we want to apply a production number to expand the non-terminal. In
this case we will consider the gene value \(\mod \maxVal\) that is for the given
non-terminal we consider the gene value modulo the maximum production number
for the given non-terminal. The second problem is what happens when we used all
the genes and we still have non-terminals to expand. Then GE proposes to start to
use the string of genes once again from the beginning.

Applying a similar technique in a TAG Evolutionary Algorithm we can construct
a derivation tree and hence a derived tree whose yield matches on a given input string.

5.1 Searching Problem Specification, Genetic Coding and De-
coding of Individuals for TAG (LTAG) Evolution

Suppose that we have a \(\text{TAG} = (N, T, I, A, S)\) and a given input string. We want to
find a derived tree that starts with \(S\) and whose yield matches a given input string.
Starting from an arbitrary \(S\)-type tree, we may apply substitutions and adjoins to
develop a derived tree. We will stop the searching process when the yield of the
derived tree matches on a given input string. As presented, the searching process is
exponential, at every step existing several possible options to chose from. In fact,
from the beginning we may chose from several \(S\)-type trees, then in the derived tree
we may chose from several nodes to apply the next derivation and once choosing a
node, we may have several possible trees to substitute or to adjoin in the given node.

Evolutionary Algorithms’ individuals represent solutions for a given problem. The
most complicated problem is to represent a derived tree in the TAG formalism using
a fixed number of genes.

Suppose that we have the \(\text{TAG} = (N, T, I, A, S)\) and there are \(|I|\) initial trees
and \(|I_S|\) initial \(S\)-type trees. We will order all the trees in the sets \(I\) and \(A\) and all the
nodes in every tree according to the node position in the string-tree representation.
Thus the tuples (tree number, node number) completely characterize all the nodes in
all the trees. We will start to build a derived tree and we will carry on in the derived
tree the nodes’ attributes such as “substitution node” or the adjoining constraints.
Now we can start to build the derived tree.

We will use the first gene $mod |I_S|$ to select the starting tree from the initial $S$-type
trees.

This initial tree will develop the derived tree.

1. We will repeat the algorithm’s steps until the length of the yield of the derived
tree will be greater or equal than the input string. If during this process we
finish the genes, we start to use the string of genes from the beginning. We count
the non-terminals that have not the $\{NA\}$ constraint in the derived tree and
let $n_{max}$ be the maximum number of the non-terminal node after the counting
operation$^3$. We will use the next gene $mod n_{max}$ to select the next node where
we will apply a derivation step

2.  
   - if the selected node is a substitution node then we count the trees that
could be substituted in our node, let $n_{s_{max}}$ be the maximum number of a
substitution tree. We will use the next gene $mod n_{s_{max}}$ to select the next
substitution tree and after performing the substitution, then we will go to
the step 1
   - if the selected node is an adjoin node then we count the trees that could
be adjoined in our node, let $n_{a_{max}}$ be the maximum number of an adjoin
tree. We will use the next gene $mod n_{a_{max}}$ to select the next adjoin tree
and after performing the adjoining, then we will go to the step 1.

We can optimize the usage of genes and whenever we have a single option for the
next operation like a single tree or a single node to choose from, we can perform the
operation without consuming the gene.

We will present the algorithm for the informal description of the genetic decoding.
We will describe only the adjoin part, the substitution being similar.

1)  
   $i = 0$ \{counter for genes index\}
2)  
   $evolvedTree = initialTrees[gene[i] \ mod \ |I_S|]$
   \{$I_S$ is the set of the initial $S$-type trees \}
3)  
   do while len(yield(evolvedTree)) < len(is) \{is : input string\}
4)  
   $i = (i + 1) \ mod \ ng$ \{ng : number of genes\}
5)  
   $n_{max} = |\{internalNodes in the evolvedTree\}| - |
   \{internalNodes with NA attribute in the evolvedTree\}|$
6)  
   $adjNode = nonTerminalCandidates[gene[i] \ mod \ n_{max}]$
7)  
   $i = (i + 1) \ mod \ ng$
8)  
   $adjSet = adjNode.Label - type auxiliary trees$
9)  
   $n_{a_{max}} = |adjSet|$
10) 
    $i = (i + 1) \ mod \ ng$
11) 
    $insertedTree = adjSet[gene[i] \ mod \ n_{a_{max}}]$

$^3$If we have Obligatory Adjoining constraints, first we should satisfy them
in the evolved tree}
{p2 is the position of the corresponding closedBracket
in the evolved tree of the adjNode}

12) \( t_1 = evolvedTree.substring(0, p_1) \)
13) \( t_2 = insertedTree.split("\ast")[0] \)
14) \{the left part of the insertedTree, \}
15) \text{until the foot symbol } "\ast" \}
16) \( t_3 = evolvedTree.substring(p_1 + 1, p_2) \)
17) \( t_4 = insertedTree.split("\ast")[1] \)
18) \{the right part of the insertedTree, \}
19) \text{after the foot symbol } "\ast" \}
20) \( t_5 = evolvedTree.substring(p_2) \)
21) \( evolvedTree = t_1 + t_2 + t_3 + t_4 + t_5 \)
22) \text{enddo}

In the “Running Examples” section we will show how the decoding function works.

5.2 Fitness Function and the Complexity of the TAG (LTAG) Evolution Algorithm

Fitness Function assign values to individuals developed by the Evolutionary Algorithm. The Fitness Function is the most important factor that directs the searching process of the Evolutionary Algorithms. Therefore a fitness function that says "yes" or "no" to the individuals of an EA is completely useless for the searching process, because the EA cannot know if a new individual is a little bit better or worse than another individual.

In our algorithm we have to encourage two aspects, the matching of characters in the input string and in the yield of the derived tree and the equal length of the two strings.

We can use several types of fitness function. The fitness function can take values on \( \mathbb{N}^3 \), first value representing the maximum length of a sequence of matched characters, the second value being the number of matches, and the third value having negative values for yields longer than the input string. When we compare different individuals during the selection process, we consider the first criterium the most important, then the second and then the third.

Let \( M \) be the number of generations after which we stop the evolution of the TAG. We assume that the crossover, mutation and selection operators have the time complexity lower than the fitness function evaluation complexity. Under these circumstances we can say that our algorithm has the time complexity \( O(M \cdot (\mu + \lambda) \cdot Time(ff)) \), where \( \mu \) is the population size in the algorithm, \( \lambda \) represents the number of children and \( Time(ff) \) is the time complexity of the fitness function. During our tests we started with \( O(n^3) \) complexity for the fitness function, then we reduced the complexity to \( O(n^2) \), but the best results we obtained with a linear fitness function.
5.3 Running Examples

We adapted the string-tree notation to simplify the internal representation of the trees. We used curly brackets to specify the constraints, rectangular brackets to specify the nodes subordinated to a nonterminal and a blank separator after terminals. We considered that nonterminals start with an uppercase character, they can be strings as long as required, eventually we include here the constraints and finally a “*” is used to indicate the end of the nonterminal representation. Inside a balanced pair (“[”, “]”) we have all the children of the nonterminal. A foot node of an auxiliary tree has no children therefore there is not necessary the pair (“[”, “]”) and instead, we will used to indicate the end of the nonterminal representation. We used curly brackets to specify the constraints, rectangular brackets to specify the nodes subordinated to a nonterminal and a blank separator after terminals. We adapted the string-tree notation to simplify the internal representation of the trees.

For tests we used the grammar TAG \( G = (N = \{ S \}, T = \{ a \}, I = \{ \alpha_1 : S\{na\}[a S[a]], \alpha_2 : S\{na\}[b S[b]] \}, A = \{ \beta_1 : S\{na\}[a S[S\{na\} * a]], \beta_2 : S\{na\}[b S[S\{na\} * b]] \}, S \) that generates \( L(G) = \{ww|w \in \{a, b\}\} \), known as the copy language. In order to explain better the decoding algorithm we will illustrate it on an example. For the grammar mentioned above and for an input string \( is = \text{“aaaabbbabaaaaababbb”} \), \( len(is) = 20 \) and the number of initial \( S - \) type trees \( |I_S| = 2 \). Suppose that we have the following string of genes: 113, 110, 248, 173, 119,... . According to the decoding algorithm instruction no. 2), gene[0]=113, the evolvedTree is \( S\{NA\}[b S[b]] \). The length of the yield of the evolvedTree is less than 20 and the algorithm will continue with the cycle from the third instruction. Next \( p_{max} \) is 1 and here due to the optimization of genes’ usage we will not increment the genes’ counter as described in the fourth instruction of the algorithm. The \( adjNode \) position is 8 (strings’ index start from 0), \( adjNode.Label \) is “S”, \( na_{max} \) is 2, gene[1]=248, the insertedTree is “S\{na\}[aS[S\{NA\}*a]]”. We have \( p_1 = 8 \), \( p_2 = 13 \), \( t_1 = “S\{NA\}[b \]”, \( t_2 = “S\{NA\}[a S\{S\{NA\}\}”, \( t_3 = “[b \]”, \( t_4 = “[a \]”, \( t_5 = “*” \), evolvedTree = “S\{NA\}[b S\{NA\}[a S[S\{NA\}[b [a [a]]]]]”. The evolution cycle continues and we get the following evolvedTrees: evolvedTree = “S\{NA\}[b S\{NA\}[a S\{S\{NA\}\][S\{NA\}[b [a [a]]]]]]”, evolvedTree = “S\{NA\}[b S\{NA\}[a S\{S\{NA\}\][S\{NA\}[S\{NA\}[b [a [a [a [a [a]]]]]]]]]]”,...

First we studied the behavior of the EA for several input examples, then we tried to compare the results with the classical parsing algorithm.

For the general behavior of the EA when solves TAG parsing, we tested two input strings, “aaaabbbabaaaaababbb” and “aaaabbbabaaaaababbaababbaabbb” having the lengths respectively 16 and 24.

We used an evolutionary algorithm with 15 individuals as the population size, each individual having 20 gene with values between 0 and 255(one byte). We could simplify the fitness function because we could stop the evolution of the derived trees as the length of the yield was equal to the length of the input string for our particular grammar. Also we considered the fitness function being the maximum length of matching characters between the input string and the yield of the derived tree considered from the beginning of the strings and from their end.
Table 4: Results of tests of EAs for TAG parsing

<table>
<thead>
<tr>
<th>gen.</th>
<th>Max</th>
<th>Average</th>
<th>Min</th>
<th>is:</th>
<th>gen.</th>
<th>Max</th>
<th>Average</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>3.27</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>2.67</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>6.27</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>6.33</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>7.13</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>6.93</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>7.40</td>
<td>6</td>
<td>3</td>
<td>12</td>
<td>8.20</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>7.87</td>
<td>7</td>
<td>4</td>
<td>12</td>
<td>8.73</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>8.67</td>
<td>7</td>
<td>5</td>
<td>12</td>
<td>9.07</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>11.20</td>
<td>8</td>
<td>6</td>
<td>12</td>
<td>9.20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>11.20</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>9.53</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>11.20</td>
<td>8</td>
<td>8</td>
<td>12</td>
<td>9.73</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>12.80</td>
<td>8</td>
<td>9</td>
<td>24</td>
<td>11.80</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>15.47</td>
<td>8</td>
<td>10</td>
<td>24</td>
<td>11.93</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

We present the results of the runs in the table 4, where *gen* represents the generation number, *is* is the input string, *Max* represents the best fitness function of an individual during one generation, *Average* is the average fitness function of individuals during one generation and *Min* is the minimum fitness function of an individual during one generation.

Theoretically the classical algorithm for parsing has the worst case complexity $O(n^6)$. The problem is that for many examples the classical algorithm does not reach the worst case and we believe that more important for compare the results for two algorithms would be to compare the results for an average behavior. In the other hand, even if the EA uses a linear fitness function, the number of generations multiplied with the number of individuals in the population could lead to an important volume of computations while solving a parsing problem.

To compare the classical parsing algorithm with the EA we used a rather empirical method to measure the number of computations. In every cycle we incremented a global variable called computations. We estimated the number of computations for both the classical algorithm and the EA using the same input string. We also could use other comparing methods such as measuring the necessary time until finding the solution, the only problem was that we implemented the algorithms in two different programming environments (VBA and Java) and the running time would have been influenced by other aspects, not only by the complexity of algorithms. Therefore we used again two input examples having the lengths of 16 ("aaaabbaaaaabba") and 20 ("aaaabbbabaaaaabbbabbb"). For the classical algorithm we needed only one run to determine the number of computations for an input example, while for the EA we considered the average result after 10 tests. The results are presented synthetically in the table 5.
Table 5: Comparative tests for classical and EA TAG parsing

<table>
<thead>
<tr>
<th></th>
<th>First example len(input)=16</th>
<th>Second example len(input)=20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computations</strong></td>
<td>classical</td>
<td>classical</td>
</tr>
<tr>
<td></td>
<td>827787</td>
<td>2153088</td>
</tr>
<tr>
<td><strong>evoAvg</strong></td>
<td>268886.3</td>
<td>661745.5</td>
</tr>
</tbody>
</table>

6 Concluding Remarks and Future Research

We proposed an Evolutionary Algorithms for Tree Adjoining Grammars parsing. We can observer that the classical parsing algorithm needs approximatively 3 times more computations than the EA to solve the same problem.

As a disadvantage of the EA parsing algorithm, for negative examples the EA will not be able to say that there is no solution. We could have some doubts that we did not let the algorithm to run enough generations, but any way, we could run tests for positive examples and we could approximate the requested number of generations required to find a solution for a certain length of the input string.

Finally, as an interesting aspect of the EA parsing algorithm, if the grammar is ambiguous, during one run for the same input string we could find different parsings as different individuals in the population.

The string-representation for trees representation could be a starting point for developing new and more efficient TAG parsing algorithms.

We believe that our algorithms are a starting point in developing new models for knowledge base representation systems, automatic text summarization etc.

For future developments, we will run the algorithms for more complex grammars including natural language parsing using the English Grammar available in the XTAG Project [9]. Also we will try guide the searching process of the EAs using some statistical information. In order to solve the negative examples problem when using EAs, we will combine the classical parsing algorithm with the EA in a concurrent manner. Then we will be able use the result of which algorithm arrives first to a conclusion. In the final phase of our research we will focus on online tests and comparative results for natural language parsing. Several theoretical results regarding the probability of not finding the solution during one generation of the EA are also expected.

Acknowledgement

We gratefully acknowledge support by the University Rovira i Virgili, within the research program "Ramon y Cajal" ref. 2002Cajal-BURV4. We also thank to James Rogers and to Giorgio Satta for their observations regarding the current paper and for their suggestions about possible future research.
References


[7] Schabes, Yves, Joshi *An Earley-Type Parser For Tree Adjoining Grammars*, Department of Computer and Information Science, University of Pennsylvania, 1988


Received: September 15, 2003
Accepted: January 3, 2004