Triangle: Language, Literature, Computation, n. 6, 2011
Publicacions Universitat Rovira i Virgili · ISSN: 2013-939X
https://revistes.urv.cat/index.php/triangle

Applications of Evolutionary Algorithms in Formal Languages

Adrian Horia Dediu

Faculty of Engineering in Foreign Languages
University "Politehnica" of Bucharest
Splaiul Independentei 313, 060042,
Bucharest, Romania

Research Group in Mathematical Linguistics
Rovira i Virgili University
Avinguda Catalunya 35, 43002,
Tarragona, Spain

E-mail: adrian.dediu@urv.cat

Summary. Starting from the model proposed by means of Grammatical Evolution, we extend the applicability of the parallel and cooperative searching processes of Evolutionary Algorithms to a new topic: Tree Adjoining Grammar parsing. We evolved derived trees using a string-tree-representation. We also used a linear matching function to compare the yield of a derived tree with a given input. The running tests presented several encouraging results. A post running analysis allowed us to propose several research directions for extending the currently known computational mechanisms in the mildly context sensitive class of languages.

1 Introduction

Evolutionary Algorithms (EAs), mainly probabilistic searching techniques, represent several converging research areas that have their roots in the 1960s. They were introduced by Hans-Paul Schwefel, Holland and De Jong. Despite the fact that various sub-domains of EAs, Evolutionary Strategies (ESs), Genetic Algorithms (GAs), Evolutionary Programming (EP), Genetic Programming (GP), etc. appeared as separate research domains they all have a basic common structure and common components. A searching space and a coding scheme representing solutions for a given problem, a fitness function, and operators to produce offspring and select a new generation are the main components of EAs. Common terms such as individuals that group together the coding scheme and the fitness function, population of individuals, and sub-populations are used in all EA sub-domains. EAs try to solve searching problems by mimicking natural principles of selection and survival of the fittest individual from a population. Real world applications of EAs deal with maximizing or minimizing objective functions such as resource location or allocation optimization. EAs are usually used for large searching space problems for which they find efficient solutions to problems that in general require a large amount of computation time. Distributed Evolutionary Algorithms (DEAs), parallel and cooperative searching processes have been proposed as an extension of centralized EAs in order to avoid premature convergence and to solve problems faster.

EAs have also been used for automatic program generation. GP, in particular, was used to generate target Lisp code. Grammatical Evolution is a new approach proposed by O'Neill in [13] that uses Context Free Grammars to automatically evolve computer programs in arbitrary languages.

Starting from the model proposed by Grammatical Evolution, we extended the applicability of the parallel and cooperative searching processes of Evolutionary Algorithms to a new topic: Tree Adjoining Grammar parsing. We evolved derived trees using a string-tree-representation. Implementing a linear matching function to compare the yield of a derived tree with a given input we obtained several encouraging results during the running tests.

Due to the high complexity of some classical parsing algorithms, long sentences analysis could be a very difficult task for a computer program. Evolutionary Algorithms used for parsing are able to process long sentences due to their reduced computational complexity. In one of our examples, we

implemented a linear complexity fitness function and in conjunction with the global complexity of the whole EA, in comparison with the $O(n^6)$ which is the complexity of the classical parsing algorithm for the same formalism, we increased the limit of the parsed words per sentence.

At the end of the paper we present a post running analysis that allowed us to propose several research directions for extening the currently known computational mechanisms in the mildly context sensitive class of languages.

2 Basic aspects of evolutionary algorithms

Various sub-domains of EAs developed a more or less rigorous theory to explain why the algorithms perform so well when they solve searching problems. For genetic algorithms there are several hypotheses that try to explain the partial results obtained. According to Goldberg [7] who introduced the schema theorem - the lower the number of symbols in the alphabet used for the coding scheme, the higher the implicit parallelism was, so an adequate coding scheme for GAs should use a binary alphabet. Studies [12] with high cardinality alphabets revealed that using or real codification had unexpected advantages(in particular, they made it possible to introduce new and stronger genetic operators such as the average crossover which performs better). In [8], Goldberg developed a new theory about how representations with a high number of symbols could perform better. Very simple fitness models are used for Evolutionary Strategies in order to obtain analytical results.

Generally, there are some unknown aspects regarding the most suitable EA operators for a given searching problem. There is a large set of recombination operators such as “one point crossover”, “two points crossover”, “shuffle crossover”, “average crossover”, “uniform mutation”, “normal mutation”, “step by step mutation”, etc. Also the select next generation operator has many variations such as: “roulette wheel selection”, “elitist selection”, “disruptive selection”, “rank space selection”, etc. We can find more details about the mentioned operators in [2], [3] and [18]. How can we know which is the best operator set capable of solving a given problem faster? How can we compare the results of different Evolutionary Algorithms when the initial population is randomly generated and the operators act randomly? Using the new paradigms of distributed Evolutionary

Algorithms, new problems arise. Do distributed EAs perform better than centralized EAs? How can we compare the performances of distributed or centralized EAs? The answers to these questions depend on the problem that EAs try to solve and up to now, due to the lack of a rigorous theory explaining the basic aspects, empirical results are the only way of shedding some light on this area.

2.1 Biological inspiration of evolutionary algorithms: common terms

Evolutionary Algorithms were inspired by biological models so the terminology used for different data structures or procedures uses the biological terms. It is somehow surprising that a list of common terms used in Evolutionary Algorithms and Genetics are explained using a single definition and not as terms with different "algorithmic meaning" or "biological meaning". The explanation relies on the fact that the common list of terms uses several basic terms such as "generation", "individual", "population", etc. the meaning of which is clear from the biological or computational points of view.

The following list presents only the most important terms used both in biology and Evolutionary Algorithms.

A *gene* is a physical and functional unit of heredity that carries information from one generation to the next.

The *chromosome* is the structure that carries the genes.

Locus means the location of a gene on a chromosome.

An *allele* represents one of the different forms of a gene that can exist at a single locus.

The *genome* is the sum of all the genetic material in a chromosome set.

The *genotype* means the specific allelic composition of a certain gene or a set of genes.

The *phenotype* is the visible or measurable characteristics of a genotype.

Mutation is a change of a gene.

Crossover means the exchange of genetic material between maternal and paternal chromosomes.

Fitness in population represents the ability of a particular genotype to reproduce itself compared to all other genotypes.

Epistasis appears when a gene expression is controlled by another gene.

2.2 Theoretical approach on evolutionary algorithms

A very good overview of Evolutionary Algorithms theory can be found in [1]. Briefly, an Evolutionary Algorithm may be defined as a 7-tuple

$$EA = (I, \Phi, \mu, \lambda, \Omega, s, t) \tag{1}$$

where:

- I represents the set of the searching space instances usually called individuals. Sometimes associated with individuals we can keep useful information for genetic operators.
- Φ is a fitness function associated with individuals.
- μ denotes the number of parents.
- λ is the number of children in the population.
- Ω is a set of genetic operators which produce new λ children when applied to the parents.
- s is the selection operator that changes the number of individuals from parents and children to produce the next generation of parents ($I^{\mu+\lambda} \rightarrow I^{\mu}$). The selection operator may also consider that after one generation the parents have completed their task and thus s selects only from the population of children ($I^{\lambda} \rightarrow I^{\mu}$).
- t represents the stop criterion which may be "Stop when a good enough value is reached by an individual fitness function", "Stop after a certain number of generations", "Stop if the population converged to a single individual", "Stop after a maximum time available for computations", etc.

A general Evolutionary Algorithm can be described as follows, where *gen* represents the generation number and s, t are, respectively, the selection operator and the termination criterion.

The structure of an **Evolutionary Algorithm** is:

gen:=0;
initialize with random values and evaluate P(0):=
$\{\langle \overrightarrow{i}_1(0), \Phi(\overrightarrow{i}_1(0))\rangle, \ldots, \langle \overrightarrow{i}_\mu(0), \Phi(\overrightarrow{i}_\mu(0))\rangle\}$;
repeat
apply genetic operators and evaluate (P(*gen*))$\rightarrow$ P'(*gen*)=
$\{\langle \overrightarrow{i}'_1(gen), \Phi(\overrightarrow{i}'_1(gen))\rangle, \ldots, \langle \overrightarrow{i}'_\lambda(gen), \Phi(\overrightarrow{i}'_\lambda(gen))\rangle\}$;
select the next generation P(*gen*+1):=s(P(*gen*),P'(*gen*));

$gen := gen+1$;
until (t(P(gen)));

2.3 Schema theorem for genetic algorithms

The schema theorem presented in [7] assumes that a genetic algorithm works with a binary coding. First, we add a new simbol * to the binary alphabet $\{0, 1\}$ which means "matches with both 0 and 1 symbols". Strings formed with symbols from the extended alphabet $\{0, 1, *\}$ are called *schemata*. A schema describes a set of binary string. As an example, the schema *10* represents the set $\{0100, 0101, 1100, 1101\}$. For a binary string of length l there are 3^l schemata and in general for strings over alphabets of cardinality k there are $(k+1)^l$ schemata.

Let us consider a schema H taken from the alphabet $\{0, 1, *\}$ whose length is l. We consider that individuals inside the algorithm have the same number of l bits. An individual in the population represents 2^l schemata while on every position we may have the actual value or *. So in a population of n individuals, there may be $n \cdot 2^l$ schemata. We can now define a schema *order*, denoted by $o(H)$ that is the number of fixed positions (characters other than the * symbol) in the schema. The *length* of a schema denoted by $\delta(H)$ is the distance between the first and the last character other than the *.

Assume that we have $A(t)$, a population A at generation t, and we use the notation $m(H,t)$ if there are m particular schema H inside the population. During the selection of a new generation a string is copied with the probability $p_i = \frac{f_i}{\sum f_j}$, where f_i is the fitness of the particular string and $\sum f_j$ is the sum of the fitness function inside the population. We write $f(H)$ the average fitness of the strings representing schema H and with $\overline{f}$ the average fitness of a population, that is $\overline{f} = \frac{\sum f_j}{n}$. We can deduce that after the selection of a new generation of n individuals the multiplicity of the schema H is: $m(H, t+1) = n \cdot p_i \cdot m(H,t)$. That is

$$m(H, t+1) = m(H,t)\frac{f(H)}{\overline{f}}$$

The formula above shows that the number of schemata that are below the average fitness decreases in the subsequent generation, and the number of schemata that are above average fitness increases during the next generation.

During the crossover operation, schema H is destroyed with the probability $p_d = \frac{\delta(H)}{l-1}$ so schema H will survive with the probability $p_s = 1 - p_d$. If we consider that for a given individual there is a crossover probability p_c then the previous formula becomes:

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l-1}$$

We used "$\geq$" instead of "=" because the schema H might appear in the population after a crossover of different individuals that do not represent the schema H.

For the mutation operator the probability of surviving for a schema H depends on every fixed surviving position. For one position the probability of surviving is $(1 - p_m)$. Therefore, for the whole schema, which has $o(H)$ fixed positions, we have a surviving probability $(1 - p_m)^{o(H)}$, which for $1 \gg p_m$ may be approximated by the expression $(1 - p_m o(H))$.

Now, considering all the effects of selection, crossover and mutation, and by multiplying the respective expressions, we obtain:

$$m(H, t+1) \geq m(H, t)\frac{f(H)}{\overline{f}} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H) \cdot p_m\right]$$

The interpretation of the above formula, known as the *schema theorem*, is: short, low order, above average schemata receive exponential surviving chances in the subsequent generations.

2.4 Distributed evolutionary algorithms

We consider the distributed EAs as 9-tuples

$$DEA = (I, \Phi, \mu, \lambda, \Omega, s, t, n, \Xi) \tag{2}$$

where:

- I, Φ, μ, λ, Ω and s are the same as for EAs.
- we consider $t = (t_f(impSol), t_e(maxGen))$ where $t_f(impSol)$ means "terminate when a solution is found that is equal to or better than impSol", and $t_e(maxGen)$ means "terminate when exceeded maxGen generations".
- n represents the number of distributed populations that evolve in parallel during the distributed EA.

- Ξ is an operator that exchanges individuals among populations.

In order to describe the structure of DEAs, we use the notation ||procP for "parallel execute procP".

The structure of our implemented
Distributed Evolutionary Algorithm is:

||initialize with random values and evaluate $P_1(0), \ldots, P_n(0)$;
$||gen_1 := 0, \ldots, gen_n := 0$;
||**repeat for every population**
||apply genetic operators and evaluate
$(P_1(gen_1)) \to P_1'(gen_1), \ldots, (P_n(gen_n)) \to P_n'(gen_n)$;
||apply exchange of individuals between populations
$\Xi(P_k, P_m)$ *for some* k, m *in* $1..n$;
||select the next generation $P_1(gen_1 + 1) := s(P_1(gen_1), P_1'(gen_1)), \ldots,$
$P_n(gen_n + 1) := s(P_n(gen_n), P_n'(gen_n))$;
$||gen_1 := gen_1 + 1, \ldots, gen_n := gen_n + 1$;
until $(\exists\, k\, s.t.\, (t_f(impSol)\ for\ P_k(gen_k))\ OR$
$(t_e(maxGen)\ for\ P_1(gen_1)\ AND \ldots t_e(maxGen)\ for\ P_n(gen_n))$;

Studying the distributed evolutionary systems we observe the existence of the following hierarchy of objects: individual, subpopulation, population, distributed population, etc.

Naturally several questions arise:

- Is this hierarchy complete or can we define more and more complex levels?
- What part of the above hierarchy might be computed in a centralized way and what part in a distributed way using multithreading, multiprocessors, computer networks?

The answer to the first question is that we can imagine defining more and more complex levels and we can use dedicated terms such as clusters of distributed populations, super clusters, etc. In order to simplify the terms used in distributed EAs, we propose the following terminology (Table 1):

We can imagine different distributed EAs computing architectures. We mention that after a centralized layer there is no point for superior layers processed in a distributed way (the gain in speed coming from the distributed processing would be canceled by the synchronization with the lower centralized levels). In Figure 1 we present the possible interactions of a

Level 0	Individual
Level 1	Subpopulation
Level 2	Population
Level 3	Distributed Population
Level 4	-
...	-
Level n	-

Table 1. Hierarchy of distributed EAs objects

centralized EA layer with other layers. We also note the the highest processing layer being responsible for the algorithm itself needs to be centralized.

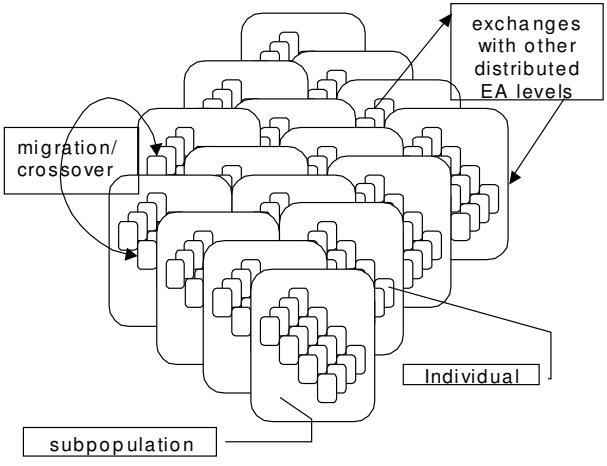

Fig. 1. Centralized EA that have connections with other distributed EA levels

Using just levels instead of hierarchical terms for the population layers we can imagine the following classes of distributed EAs (Table 2). (Table 2):

Based on the previous considerations we introduce a new notation for distributed EA classes. We denote by $\langle n, m \rangle$, the class of a distributed EA, where n is the total number of the EA layers, and m is the number of the

Level 0	Level 1	Level 2	Level 3	Level 4	...	Level n
Distributed	Distributed	Distributed	Distributed	Distributed	Distributed	Distributed
Centralized	Distributed	Distributed	Distributed	Distributed	Distributed	Distributed
	Centralized	Distributed	Distributed	Distributed	Distributed	Distributed
		Centralized	Distributed	Distributed	Distributed	Distributed
			Centralized	Distributed	Distributed	Distributed
...	...	...	...	...	...	...
					Centralized	Distributed

Table 2. Possible distributed EAs computing architectures

distributed levels. As an example, the distributed GA implementation where individuals are decentralized objects belongs to class $\langle 1, 0\rangle$.

Usual distributed EAs implementations deal with $\langle n, 3\rangle$ classes. Figure 1 is also an example of such distributed EA architecture.

2.5 Evolutionary algorithms used in problem optimization, first example

When we try to solve a problem using EAs, we should first design the search space. Depending on the number of dimensions and the searching precision we automatically get the genetic structure of individuals. In almost all cases, individuals in an EA represent solutions for the given problem and the fitness function evaluates how good the solution is. Finally, choosing the set of control parameters is a matter of personal experience and intuition.

The game of TicTacToe is played between two partners on a board as we can see in Figure 2. Every partner marks the squares of the board with distinct symbols. The goal of the game is to obtain a line (horizontal, vertical or diagonal) of a certain length marked with the same symbol. In our implementation one of the partners is an Evolutionary Algorithm that automatically proposes the next move.

In the case of the TicTacToe game, what would be a solution for a given situation on the board? We consider that individuals are formed from 2 genes, a starting coordinate and a direction to follow. How would one individual be evaluated with respect to the proposed coding? Suppose that EA plays with X. The value of an individual is the sum of already marked X values. We assign negative values for 0 symbols in the individual's "body". In order to increase the "intelligence" of the proposed solution we considered not the best individual as the next move, but the intersection of several best individuals proposed by the algorithm.

More details regarding the implementation are not within the scope of the current article. The implementation of the EA that plays TicTacToe

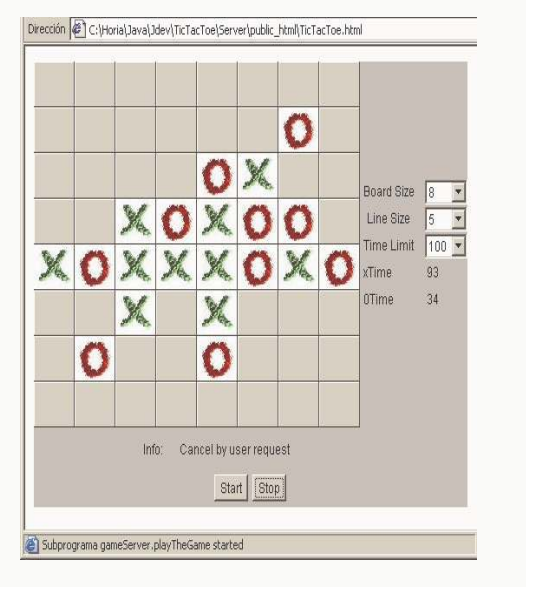

Fig. 2. An implementation of a TicTacToe game partner using an EA engine

may be tested on the web address `http://grammars.grlmc.com/GRLMC/PersonalPages/AdrianHoria/tictactoe.html`.

2.6 Grammatical evolution

Grammatical Evolution is a new approach proposed by O'Neill in [13] that uses Context Free Grammars and Genetic Algorithms to automatically evolve computer programs in arbitrary languages. Despite the fact that the goal sounds very ambitious, the implementation has several limitations.

In paper [13] the EAs find a function of one independent variable and one dependent variable in symbolic form, which fits a given sample of 20 data points (x_i, y_i). The quadratic polynomial function $x^4 + x^3 + x^2 + x$ with points from the interval $[-1, 1]$ was used.

The grammar used was the following:

$$
\begin{aligned}
1) < expr > \quad &::= < expr >< op >< expr > \quad (0\\
&\quad | \ (< expr >< op >< expr >) \ (1\\
&\quad | < pre-op > (< expr >) \quad (2\\
&\quad | < var > \quad (3\\
2) < op > \quad &::= +|-|/|*\\
3) < pre-op > &::= sin|cos|tan|log\\
4) < var > \quad &::= X.
\end{aligned}
$$

The algorithm constructs a symbolic expression using the sentential form. The *genetic coding* is a string of bytes. First the algorithm starts with the starting symbol ($< expr >$) and it expands the leftmost symbol considering the gene value mod the number of choices. Then the next leftmost symbol and the next gene are used. If there is only one choice then the symbol is expanded without considering the gene value. This procedure continues until all the nonterminals in the sentential form were expanded. If the string of genes is exhausted before the nonterminals in the sentential form, then the string of genes is used once again from the beginning as if it were a circular string.

The *fitness function* evaluation promotes a multicriterial optimization, which maximizes the number of fitting points and minimizes the error

$$\sum_{i=1}^{20} |f(x_i) - y_i|.$$

This approach is also useful in parsing . We can observe that GE cannot really evolve programs, only functions specified by samples.

3 Evolutionary algorithms for tree adjoining grammatical evolution

Context-free grammars (CFGs) are a well known class of grammars that are extensively used for programming languages and they can also describe almost all structures of natural languages. Yet in shuch cases as multiple agreement languages $\{a_1^n a_2^n \dots a_k^n | n \geq 1, k \geq 3\}$, copy languages $\{ww | w \in \{a,b\}^*\}$ and cross agreement $\{a^n b^m c^n d^m | n, m \geq 1\}$, context-free grammars are not the most appropriate investigation instrument for natural language 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) are an important class of grammars, originally motivated by linguistic considerations, which subsequently yielded important mathematical and computational results, which in turn had linguistic implications. A.K. Joshi and Y. Shabes published an overview of TAGs in [11].

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

A *recognizer* is an algorithm that takes a string as input and either accepts it or rejects it, depending on whether it belongs to the language of a grammar or not. In Figure 3 we can see a recognizer data flow.

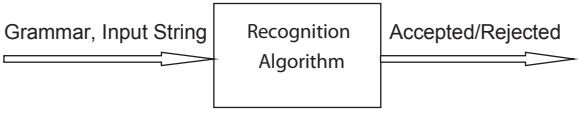

Fig. 3. A recognizer data flow

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

3.1 Definition, components and composition operations in tree adjoining grammars

Definition 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 $I \bigcup A$ are called *elementary trees*.

The trees with roots labeled by the nonterminal A are called *A-type trees*.

We can associate a *node address* (Gorn-position) with every node in a tree 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 and then the number of children counted from the left. In the set of elementary trees we can form a global address using tuples formed by (*treeName*, *nodeAddress*).

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

TAGs operate with two composition operations, *adjoining* and *substitution*.

Definition 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 3. *Adjoining builds a new tree from an auxiliary tree β and another tree α, which can be initial, auxiliary or a derived tree. Let us consider that the root node of the auxiliary tree β is labeled by X (also the foot node, by definition). If an internal node of the tree α is labeled by X then the adjoining operation will construct a tree as follows:*

- *the sub-tree of α dominated by X, call it t, is removed from the tree α, leaving a copy of X behind.*

- *the auxiliary tree β is attached to the copy of the node labeled by X in the excised tree α.*
- *the sub-tree t is attached to the foot node of β and the root of t is identified with the foot node of β (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 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 $SA(AT)$ 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 NA forbids any adjunction on a given node. We have NA=$SA(\emptyset)$.*
- *Obligatory adjunction written 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 adjunction or the substitution operations we obtain the *derived trees*. They do not have the information about how they were built, so we need to build a special structure that can specify how a derived tree was constructed.

Definition 5. *A TAG derivation tree is a tree used to show how a derived tree was constructed and it has:*

- *a 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 its frontier and if all the obligatory adjunction constraints 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:$L(G) = \{w | w = yield(t), t \in T_G\}$

3.2 Lexicalized grammars

Lexicalized grammars, presented in [11] by Joshi and Shabes, have both linguistic and formal importance.

Definition 6. *A grammar is lexicalized if it consists 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 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 2. *It is decidable whether or not a string is accepted by a lexicalized grammar.*

Definition 7. *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 a lexical item on the right hand side and sometimes CFGs can be infinitely ambiguous, containing recursive derivation chains such as $X \Longrightarrow^* X$. Lexicalization of finitely ambiguous CFGs achieved by transforming them into Greibach Normal Form [5], can be regarded as *weak lexicalization* since we do not preserve also the structure of the derivation trees. The above definition of lexicalization may be regarded as *strong lexicalization*.

Definition 8. *A tree-substitution grammar (TSG) is a TAG without auxiliary trees and without the adjoin operation.*

We use the same terminology for *derived trees* and *derivation trees*. A tree is *completed* if it has only terminals on its frontier. We 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 [11].

Proposition 3. *Finitely ambiguous CFGs cannot be lexicalized by a TSG.*

Proposition 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.

3.3 A classical tree adjoining grammar recognizer

We present a version of an Earley algorithm for TAGs described by Joshi and Schabes in [11]. 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-below, right-and-below, right-and-above. Because of this representation, there might be several equivalent items in the chart with different representation as we can see in Figure 4, for every dotted tree there can be two equivalent items in the chart whit right-and-above and right-and-below equivalent dot positions on the left picture and right-and-above and left-and-above equivalent dot positions on the right picture.

We introduce a shorthand notation for tree structures namely, string representation for trees, in order to simplify tree descriptions. For the definitions and notations related to N, T-trees we refer the reader to [6]. An N, T-tree can be represented as a string by the mapping

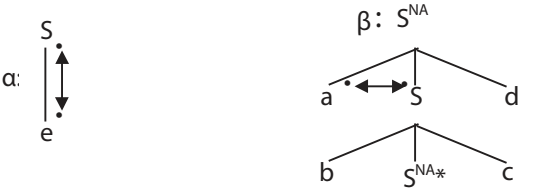

Fig. 4. Equivalent dot positions

$$
\begin{aligned}
str \;:\; \mathcal{T}_N(T) &\to (N_B \cup T)^*, \text{ defined as} \\
str(t_a()) &= a,\ a \in T \cup \{\lambda\} \text{ (a single-node tree labeled } a), \\
str(t_A(t_1, \cdots, t_m)) &= A\, str(t_1) \cdots str(t_m)\, \bar{A},\ A \in N \\
&\quad \text{(a tree labeled } A \text{ with subtrees } t_1, \cdots, t_m).
\end{aligned}
$$

where $N_B = N \cup \bar{N}$, $\bar{N} = \{\bar{A} \mid A \in N\}$.

If we can rapidly distinguish the nonterminals from the terminals by using, respectively, uppercase letters and lowercase letters from the alphabet, then we can simplify the notation even more, and use only closed brackets instead of bar symbols : i.e. $str(t_A(t_1, \cdots, t_m)) = A\, str(t_1) \cdots str(t_m)\,]$, $A \in N$.

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. Additionally, the problem of equivalent dotted items disappears, as we can see the trees in the Figure 4 have the following representation: $Se.]$ and $S^{NA}a.SbS^{NA}*]c]d]$.

Informal description of a TAG (LTAG) recognizer

In an attempt 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,

$$s = [treeName, dottedStringTree, i, j, k, l, sat?]$$

where:

- *treeName* is the name of an elementary tree.
- *dottedStringTree* is the string-representation of the tree *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.
- *sat*? is used to disallow more than one adjunction in the same node. *sat*? takes values ranging from $\{nil, true\}$, where *nil* means no adjunction has yet been performed on the dotted node and *true* means that an adjunction has been 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 α_i, chart *C* contains all items of the form:

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

Depending on the items in chart *C*, new items are added to the chart. According to the items in the chart four basic operations add new items: PREDICT, SCAN, COMPLETE and ADJOIN to which work basically as described in [11]. 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:

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

where α 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

Formal description of a TAG (LTAG) recognizer

We use the index in the string-tree representation as the nodes' adresses and the function *pos* to return the node number in the string-tree representation.

We use the following notations: *Var* is the set of variables; *Term* is the set of terminals; *StartingSymbol* is the value of the starting symbol; *trees* is the set of all elementary trees; for a given tree t we write $str(t)$

Functional description	After dot	Current item	Supplementary conditions
Scan 1	a	(t1, s1+"$.a$"+s2, i, j, k, l, nil)	inputString[l+1]=a
Added item	(t1, s1+"a."+s2, i, j, k, l+1, nil)		
Scan 2	λ	(t1, s1+"."+λ+s2, i, j, k, l, nil)	
Added item	(t1, s1+λ+"."+s2, i, j, k, l, nil)		
Predict 1	V	(t1, s1+"."+V+s2, i, j, k, l, nil),	for each $at \in Adj(t1, pos(V))$
Added items	(at, "."+$str(at)$, l, -1, -1, l, nil)		
Predict 2	V	(t1, s1+"."+V+s2, i, j, k, l, nil),	$OA(V) = false$
Added item	t1, s1+V+"."+s2, l, -1, -1, l, nil		
Predict 3	*	(t1, s1+V+".*"+s2, l, -1, -1, l, nil)	for each t in $trees$, {for each V in $str(t)$, { str(t)=s3+V+s4 $t1 \in Adj(t, pos(V))$ }}
Added items	(t, s3+V+"."+s4, l, -1, -1, l, nil)		
Complete 1	]	(t1, s1+".]"+s2, i, j, k, l, nil)	for each it in C it=(t2, s3+V+".*]"+s4, i, -1, -1, i, nil), $\hookleftarrow$(s1+".]")=V
Added items	(t2, s3+V+"*.]"+s4, i, i, l, l, nil)		
Complete 2	]	(t1, s1+".]"+s2, i, j, k, l, nil)	for each it in C it=(t1, s3+"."+V+s4, h, j', k', i, sat?) $\hookleftarrow$(s1+".]")=V
Added items	(t1, s1+"]."+s2, h, max(j, j'), max(k, k'), l, sat?)		
Adjoin		(t1, V+s1+"*"+s2+".", i, j, k, l, nil)	for each it in C it=(t2, s3+".]"+s4, j, p, q, k, nil) $\hookleftarrow$(s3+".]")=V, $t1 \in Adj(t2, pos(V))$
Added items	(t2, s3+"]."+s4, i, p, q, l, true)		

Table 3. New items added to the chart

the string-tree representation of the tree t[1]; *NullAdjoining* is the set of pairs (*treeName*, *nodeAddress*) for which the *NullAdjoining* attribute is set; $Adj(treeName, nodeAddress)$ is the set of auxiliary trees that can be adjoined in the *treeName* at the address *nodeNumber*; *inputString* is the value of the string to be recognized and we have the $inputString = a_1 \ldots a_n$; *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; $\max(i, j)$ is the maximum value between i and j; *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:

```
counter =1
Starting from the dot position,
 go right in the dotted string-tree,
        increment the counter for any ''['',
        decrement the counter for any V in Var,
if counter=0 then return V
```

Internally the algorithm builds the *initialTrees* and *auxiliaryTrees* sets based on the *trees* set. We also use the *item* definition as described in the informal description. The recognition algorithm works with a set of items collected in a chart C. Also we define a procedure *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:

```
function recognize
   C = φ
   for each tree in initialTrees
      if root(tree)=StartingSymbol then
         chartAdd([tree,''.''+stringDescription(tree),0,−1,−1,0,nil])
      end if
   next tree

   apply for each item in C
      if (addNewItems(item)=''stop'') then
         return(''recognized'')
      end if
   until no more items are added in the chart C
```

[1] We can distinguish between initial trees and auxiliary trees, because auxiliary trees will contain the footer symbol "*"

```
    return(''not recognized'')
end function
```

The function *addNewItems*(*currentItem*) returns "stop" if a new item of the form:

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

where α is an S-type initial tree is to be added to chart C.

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

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

A step by step example for the classical TAG (LTAG) recognizer

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

Example 1.

Table 4 describes the TAG $G_2 = (N = \{S\}, T = \{a, b, c, d, e\}, I = \{\alpha : Se]\}, A = \{\beta : S^{NA}aSbS^{NA}*]c]d]\}, S)$.

Variable name	Type	Value
Var	set	{S}
Term	set	{a,b,c,d,e}
StartingSymbol	value	S
trees	α	Se]
trees	β	SaSbS*]c]d]
NullAdjoining	β	1,5
ObligatoryAdjoining	set	{}
inputString	value	aabbeccdd

Table 4. Recognizer description of the TAG G_2

The recognizer fills in the chart as described in Table 5.

Input read	Item			Apply
	No	from	Value	
The positions in the input string are: ${}_0a_1a_2b_3b_4e_5c_6c_7d_8d_9$				
	1	init	α, .Se], 0, -1, -1, 0, n	P1+2:P2+3
	2	P1(1)	β, .S^{NA}aSbS^{NA}*]c]d], 0, -1, -1, 0, n	P2+4
	3	P2(1)	α, S.e], 0, -1, -1, 0, n	S1-
	4	P2(2)	β, S^{NA}.aSbS^{NA}*]c]d], 0, -1, -1, 0, n	S1+5
a	5	S1(4)	β, S^{NA}a.SbS^{NA}*]c]d], 0, -1, -1, 1, n	P1+6:P2+7
a	6	P1(5)	β, .S^{NA}aSbS^{NA}*]c]d], 1, -1, -1, 1, n	P2+8
a	7	P2(5)	β, S^{NA}aS.bS^{NA}*]c]d], 1, -1, -1, 1, n	S1-
a	8	P2(6)	β, S^{NA}.aSbS^{NA}*]c]d], 1, -1, -1, 1, n	S1+9
aa	9	S1(8)	β, S^{NA}a.SbS^{NA}*]c]d], 1, -1, -1, 2, n	P1+10:P2+11
aa	10	P1(9)	β, .S^{NA}aSbS^{NA}*]c]d], 2, -1, -1, 2, n	P2+12
aa	11	P2(9)	β, S^{NA}aS.bS^{NA}*]c]d], 2, -1, -1, 2, n	S1+13
aa	12	P2(10)	β, S^{NA}.aSbS^{NA}*]c]d], 2, -1, -1, 2, n	S1-
aab	13	S1(11)	β, S^{NA}aSb.S^{NA}*]c]d], 2, -1, -1, 3, n	P2+14
aab	14	P2(13)	β, S^{NA}aSbS^{NA}.*]c]d], 3, -1, -1, 3, n	P3+15:P3+16
aab	15	P3(14)	α, S.e], 3, -1, -1, 3, n	S1-
aab	16	P3(14)	β, S^{NA}aS.bS^{NA}*]c]d], 3, -1, -1, 3, n	S1+17
aabb	17	S1(16)	β, S^{NA}aSb.S^{NA}*]c]d], 3, -1, -1, 4, n	P2+18
aabb	18	P2(17)	β, S^{NA}aSbS^{NA}.*]c]d], 4, -1, -1, 4, n	P3+19:P3+20
aabb	19	P3(18)	α, S.e], 4, -1, -1, 4, n	S1+21
aabb	20	P3(18)	β, S^{NA}aS.bS^{NA}*]c]d], 4, -1, -1, 4, n	S1-
aabbe	21	S1(19)	α, Se.], 4, -1, -1, 5, n	C1(21+18)+22
aabbe	22	C1(21,18)	β, S^{NA}aSbS^{NA}*.]c]d], 4, 4, 5, 5, n	C2(22+17)+23
aabbe	23	C2(22,17)	β, S^{NA}aSbS^{NA}*].c]d], 3, 4, 5, 5, n	S1+24
aabbec	24	S1(23)	β, S^{NA}aSbS^{NA}*]c.]d], 3, 4, 5, 6, n	C1(24+14)+25
aabbec	25	C1(24,14)	β, S^{NA}aSbS^{NA}*.]c]d], 3, 3, 6, 6, n	C2(25+13)+26
aabbec	26	C2(25,13)	β, S^{NA}aSbS^{NA}*].c]d], 2, 3, 6, 6, n	S1+27
aabbecc	27	S1(26)	β, S^{NA}aSbS^{NA}*]c.]d], 2, 3, 6, 7, n	C2(27+9)+28
aabbecc	28	C2(27,9)	β, S^{NA}aSbS^{NA}*]c].d], 1, 3, 6, 7, n	S1+29
aabbeccd	29	S1(28)	β, S^{NA}aSbS^{NA}*]c]d.], 1, 3, 6, 8, n	C2(29+6)+30
aabbeccd	30	C2(29,6)	β, S^{NA}aSbS^{NA}*]c]d]., 1, 3, 6, 8, n	A(30+24)+31
aabbeccd	31	A(30,24)	β, S^{NA}aSbS^{NA}*]c.]d], 1, 4, 5, 8, t	C2(31+5)+32
aabbeccd	32	C2(31,5)	β, S^{NA}aSbS^{NA}*]c].d], 0, 4, 5, 8, n	S1+33
aabbeccdd	33	S1(32)	β, S^{NA}aSbS^{NA}*]c]d.], 0, 4, 5, 9, n	C2(33+2)+34
aabbeccdd	34	C2(33,2)	β, S^{NA}aSbS^{NA}*]c]d]., 0, 4, 5, 9, n	A(34+21)+35
aabbeccdd	35	A(34,21)	α, Se.], 0, -1, -1, 9, t	C2(35+1)+36
aabbeccdd	36	C2(35,1)	α, Se]., 0, -1, -1, 9, n	recognized

Table 5. Chart C after the recognition of the TAG G_2 and the input string aabbeccdd

Complexity considerations for the TAG (LTAG) recognizer algorithm

The presented algorithm has the worst case time complexity $O(|A| \cdot |A \bigcup I| \cdot M \cdot n^6)$ where $|A|$ is the number of auxiliary trees, $|A \bigcup I|$ 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 so we can dramatically reduce the number of elementary trees used in the parsing process. The case complexity is worst during the ADJOIN operation. As we can observe in Table 3, during the adjoin operation we have to combine two items ($t1$, ..., i, j, k, l, nil) and ($t2$, ..., j, p, q, k, nil). Therefore we have at most n^6 instances of indices (i, j, k, l, p, q) and we can call the adjoin operation $|A| \cdot |A \bigcup I| \cdot M$ times.

3.4 Evolutionary algorithm for tree adjoining grammar parsing

We can find an introduction to 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 [13], 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.

GE solves two main problems. 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 consider the gene value *mod maxValidValue*: 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.

Basic aspects of TAG (LTAG) evolution

Suppose that we have a $TAG = (N, T, I, A, S)$ 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 stop the searching process when the yield of the derived tree matches on a given input string. As presented, the searching process is exponential, and at every step there are several possible options to choose from. In fact, from the beginning we may choose from several S-type trees, so in the derived tree we may choose from several nodes to apply the next derivation and once a node has been chosen, 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 $TAG = (N, T, I, A, S)$ and there are $|I|$ initial trees and $|I_S|$ initial S-type trees. We 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 start to build a derived tree and we 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 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 repeat the algorithm's steps until the length of the yield of the derived tree will be greater than or equal to the input string. If we finish the genes during this process, we start to use the string of genes from the beginning again. We count the non-terminals that do not have the {NA} constraint in the derived tree and let n_{max} be the maximum number of the non-terminal node after the counting operation[2]. We use the next gene mod n_{max} to select the next node where we 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 ns_{max} be the maximum number of a substitution tree. We use the next gene mod ns_{max} to select the next substitution tree and after performing the substitution, we go to the step 1

[2] If we have Obligatory Adjoining constraints, they must be satisfied first.

- If the selected node is an adjoin node then we count the trees that could be adjoined in our node, let na_{max} be the maximum number of an adjoin tree. We use the next gene mod na_{max} to select the next adjoin tree and after performing the adjoining, we go to 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 present the algorithm that describes the genetic decoding. We give only the adjoin part, because the substitution is 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)       na_max = |adjSet|
10)      i = (i+1) mod ng
11)      insertedTree = adjSet[gene[i] mod na_max]
       {p1 is the position of the adjNode
          in the evolved tree}
       {p2 is the position of the corresponding
          closedBracket in the evolved tree of the adjNode}
12)      t1 = evolvedTree.substring(0, p1)
13)      t2 = insertedTree.split("*")[0]
14)      {the left part of the insertedTree,
15)         until the foot symbol "*"}
16)      t3 = evolvedTree.substring(p1 + 1, p2)
17)      t4 = insertedTree.split("*")[1]
18)      {the right part of the insertedTree,
19)         after the foot symbol "*"}
20)      t5 = evolvedTree.substring(p2)
21)      evolvedTree = t1 + t2 + t3 + t4 + t5
22) enddo
```

In the "Running Examples" section we show how the decoding function works.

Fitness function complexity

The fitness function assigns values to individuals developed by the Evolutionary Algorithm. It 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 the characters in the input string and in the yield of the derived tree must match and the two strings must be of equal length.

We can use several types of fitness function. The fitness function can take values on $\mathbf{N^3}$, the 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, the first criterion is 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 time complexity of the crossover, mutation and selection operators is less than the fitness function evaluation complexity. Under these circumstances we can say that our algorithm has a time complexity of $O(M \cdot (\mu + \lambda) \cdot Time(ff))$, where μ is the population size in the algorithm, λ 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, which we then reduced to $O(n^2)$. But the results were best with a linear fitness function.

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, the strings can be as long as required, constraints are included 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 so the pair("[", "]") is not necessary and instead, we have only the foot marker that is the "*".

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 the decoding algorithm better we illustrate it on an example. For the grammar mentioned above and for an input string *is* ="aaaabbbabbaaaabbbabb", $len(is)$ 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 n_{max} is 1 and here due to the optimization of genes' usage we do not increment the gene counter as described in the fourth instruction of the algorithm. The *adjNode* position is 8 (string index starts from 0), $adjNode.Label$ is "S", na_{max} is 2, gene[1]=248, the *insertedTree* is "S{na}[aS[S{NA}*a]]". We have $p1 = 8$, $p2 = 13$, $t1 =$ "$S\{NA\}[b\ $", $t2 =$ "$S\{NA\}[a\ S[S\{NA\}$", $t3 =$ "$[b\ ]$", $t4 =$ "$a\ ]]$", $t5 =$ "$]$", $evolvedTree =$ "$S\{NA\}[b\ S\{NA\}[a\ S[S\{NA\}[b\ ]a]]]$".

The evolution cycle continues and we get the following *evolvedTrees*:
$evolvedTree =$ "$S\{NA\}[b\ S\{NA\}[a\ S\{NA\}[a\ S[S\{NA\}[S\{NA\}[b\ ]a\ ]a\ ]]]]$",
$evolvedTree =$ "$S\{NA\}[b\ S\{NA\}[a\ S\{NA\}[aS\{NA\}[b\ S[S\{NA\}\ [S\{NA\}\ [S\{NA\}\ [b\ ]a\ ]a\ ]b\ ]]]]]$",...,
$evolvedTree =$ "$S\{NA\}[b\ S\{NA\}[a\ S\{NA\}[aS\{NA\}[b\ S\{NA\}[b\ S\{NA\}[bS\{NA\}\ [a\ S\{NA\}[aS\{NA\}[a\ S\{NA\}[aS[S\{NA\}[S\{NA\}[S\{NA\}[S\{NA\}\ [S\{NA\}[S\{NA\}[S\{NA\}[S\{NA\}[S\{NA\}[b\ ]a\ ]a\ ]b\ ]b\ ]\ b\ ]a\ ]a\ ]a\ ]a\ ]]]]]]]]]]$".

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

For the general behavior of the EA when it solves TAG parsing, we tested two input strings, "aaaabbbaaaaabbba" and "aaaabbbaabbaaaaabbbaabba" with lengths 16 and 24, respectively.

We used an evolutionary algorithm with 15 individuals as the population size. Each individual had 20 genes with values between 0 and 255 (one byte). We managed to simplify the fitness function because we stopped 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. We also considered

Input string							
aaaabbbaaaaabbba				**aaaabbbaabbaaaaabbbaabba**			
gen.	**Max**	**Average**	**Min**	**gen.**	**Max**	**Average**	**Min**
0	7	3.27	2	0	5	2.67	1
1	7	6.27	6	1	8	6.33	5
2	16	7.13	6	2	8	6.93	6
3	16	7.40	6	3	12	8.20	7
4	16	7.87	7	4	12	8.73	7
5	16	8.67	7	5	12	9.07	8
6	16	11.20	8	6	12	9.20	8
7	16	11.20	8	7	12	9.53	9
8	16	11.20	8	8	12	9.73	9
9	16	12.80	8	9	24	11.80	9
10	16	15.47	8	10	24	11.93	9

Table 6. Results of tests of EAs for TAG parsing

the fitness function to be the maximum length of matching characters between the input string and the yield of the derived tree considered from the beginning and the end of the strings.

We present the results of the runs in table 6, 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 than comparing the results for two algorithms would be to compare the results for an average behavior. On the other hand, even if the EA uses a linear fitness function, the number of generations multiplied by the number of individuals in the population could lead to a considerable volume of computations while solving a parsing problem.

To compare the classical parsing algorithm with the EA we used a an empirical method to measure the number of computations. In every cycle we incremented global variables called computations. We estimated the number of computations for both the classical algorithm and the EA using the same input string. We also used other comparative methods such as measuring the time it took to find 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 again used two input examples whit lengths of 16 (“aaaabbbaaaaabbba”) and 20 (“aaaabbbabbaaaabbbabb”). For the classical algorithm we needed only one run to determine the number of computations for an input example, while for EA we considered the average result after 10 tests. The results are presented synthetically in table 7.

	First example len(input)=16	Second example len(input)=20
	Computations	**Computations**
classical	827787.0	2153088.0
evoAvg	268886.3	661745.5

Table 7. Comparative tests for classical and EA TAG parsing

4 Post running analysis

Let us recall steps 12) to 21) from the algorithmic description of the adjoin operation.

12) $t1 = evolvedTree.substring(0, p1)$
13) $t2 = insertedTree.split(“*”)[0]$
14) {the left part of the *insertedTree*,
15) until the foot symbol “*”}
16) $t3 = evolvedTree.substring(p1 + 1, p2)$
17) $t4 = insertedTree.split(“*”)[1]$
18) {the right part of the *insertedTree*,
19) after the foot symbol “*”}
20) $t5 = evolvedTree.substring(p2)$
21) $evolvedTree = t1 + t2 + t3 + t4 + t5$

We can see in Figure 5 that the adjoin operation is similar to something like a synchronized pumping of two strings, the left side and the right side of an auxiliary tree with respect to the foot.

We also know that TAGs have some limitations in the sense that the language $a^n b^n c^n d^n e^n$ is not a TAL. Considering the previous observations, we define two formalisms that can extend the TAG formalism.

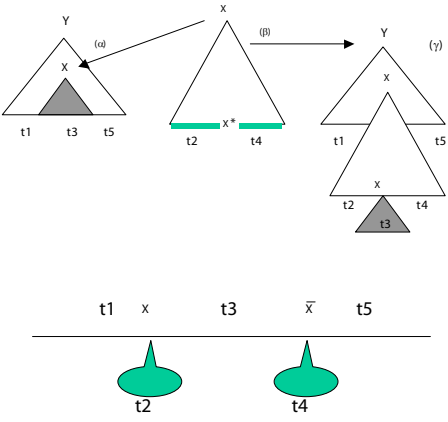

Fig. 5. Synchronization in TAG formalism

Notations: $N_B = N \cup \bar{N}$, $\bar{N} = \{\bar{A} \mid A \in N\}$, $D_{N,T}$ is the Dyck language over N, "enriched" with terminals (see its grammar G below), which we call *bracketed language*.

$$G = (S, N_B \cup T, P, S),$$
$$P = \{S \to \lambda, S \to SS\} \cup \{S \to a \mid a \in T\} \cup \{S \to aS\bar{a} \mid a \in N\}.$$

Definition 9. *A* step-synchronized rewriting system *(SSR) is a quadruple* $G = (N, T, P, S)$, *where* N *is a finite set of non-terminal symbols,* T *is a finite set of terminal symbols* ($N \cap T = \emptyset$), $S \in N$ *is a starting symbol, and* P *is a finite set of rules that are tuples of context-free productions.*

Example 2. Consider the following system

$$\begin{aligned} G_1 = (\{S, A, B, C, D\}, \{a, b, c, d, e, f, g, h\}, P, S), \\ \text{where } P = \{(S \to ABCD), \\ (A \to aAb, C \to cCd), (A \to ab, C \to cd), \\ (B \to eBf, D \to gDh), (B \to ef, D \to gh)\}. \end{aligned}$$

For any $i \geq 0$ we define the sets $N_i = \{A^{(i)} \mid A \in N\}$ (intuitively, symbols added at step i of a derivation), $V_i = N_i \cup T$, and a homomorphism

$h_i : V \to V_i$ defined by $h_i(a) = a$ for all $a \in T$ and $h_i(A) = A^{(i)}$ for all $A \in N$. We also define $N_* = \{A_i \mid A \in N, i \geq 0\}$ and $V_* = N_* \cup T$.

Configurations of the systems G are represented by a string in V_*^* and a number. The starting configuration of G is $(S^{(0)}, 0)$. We say that a configuration (w, j) directly derives (w', j') (denoted as $\Rightarrow$) iff $j' = j + 1$ and w and w' can be represented in the following way:

$$\begin{aligned} w &= w_1 A_1^{(i)} w_2 \cdots w_k A_k^{(i)} w_{k+1}, \\ w' &= w_1 h_{j+1}(x_1) w_2 \cdots w_k h_{j+1}(x_k) w_{k+1}, \\ &\quad (A_1 \to x_1, \cdots, A_k \to x_k) \in P. \end{aligned}$$

The derivation relation ($\Rightarrow^*$) is defined as a reflexive and transitive closure of $\Rightarrow$. The generated language is defined as $L(G) = \{w \in T^* \mid (S^{(0)}, 0) \Rightarrow^* (w, n)\}$.

Example 3. For the system G_1, consider the derivation:
$(S^{(0)}, 0) \Rightarrow (A^{(1)}B^{(1)}C^{(1)}D^{(1)}, 1) \Rightarrow^*$
$(a^m A^{(m+1)} b^m B^{(1)} c^m C^{(m+1)} d^m D^{(1)}, m+1) \Rightarrow^*$
$(a^m A^{(m+1)} b^m e^n B^{(m+n+1)} f^n c^m C^{(m+1)} d^m g^n D^{(m+n+1)} h^n, m+n+1) \Rightarrow$
$(a^{m+1} b^{m+1} e^n B^{(m+n+1)} f^n c^{m+1} d^{m+1} g^n D^{(m+n+1)} h^n, m+n+2) \Rightarrow$
$(a^{m+1} b^{m+1} e^{n+1} f^{n+1} c^{m+1} d^{m+1} g^{n+1} h^{n+1}, m+n+3)$ (see Figure 6).

The synchronization relation is represented by connecting the synchronized nonterminals in the sentential form.

$$(a^m \overbrace{A^{(m+1)} b^m e^n \underbrace{B^{(m+n+1)}}} f^n c^m \overbrace{C^{(m+1)}} d^m g^n \underbrace{D^{(m+n+1)}} h^n, m+n+1)$$

Fig. 6. For Example 3, the synchronization is by the derivation step.

We define $SSRS_k$ as the set of all step-synchronized rewriting systems $G = (N, T, P, S)$, where $P \subseteq (N \times V^*)^{\leq k}$ (at most k productions in all rules). We write $SSRS_* = \bigcup_{k \geq 1} SSRS_k$.
Observation: $L(SSRS_1) = CF$.
Observation: $\{a_1^n \cdots a_{2k}^n \mid n \geq 1\} \in L(SSRS_k)$.

Definition 10. *A* bracketed-synchronized rewriting system (BSR) *is a 4-tuple* $G = (N_B, T, P, S)$, *where* N_B *is a finite set of bracketed non-terminal symbols,* T *is a finite set of terminal symbols* ($N_B \cap T = \emptyset$), $S \subseteq D_{N,T}$ *is a finite set of starting*

axioms and P is a finite set of rules that are pairs of context-free productions of the form $(A \to w_1, \bar{A} \to w_2)$, *where* $w_1 w_2 \in D_{N,T}$.

We say that w directly derives in w' (denoted as $w \Rightarrow w'$) iff w and w' allow representation:

$$w = w_1 A w_2 \bar{A} w_3, \quad w_2 \in D_{N,T},$$
$$w' = w_1 x_1 w_2 x_2 w_3, \quad (A \to x_1, \bar{A} \to x_2) \in P.$$

Observation: In each derivation starting from an axiom, every sentential form is in $D_{N,T}$. The derivation relation and the language are defined as usual. *BSRS* is the set of BSR systems.

Example 4. For the system $G_2 = (\{A, \bar{A}\}, \{a, b, c, d\}, P, AA\bar{A}\bar{A}), \ P = \{(A \to aAb, \bar{A} \to c\bar{A}d), (A \to ab, \bar{A} \to cd)\}$, consider the following derivation:
$AA\bar{A}\bar{A} \Rightarrow^* Aa^{n-1}Ab^{n-1}c^{n-1}\bar{A}d^{n-1}\bar{A} \Rightarrow^*$
$a^{m-1}Ab^{m-1}a^{n-1}Ab^{n-1}c^{n-1}\bar{A}d^{n-1}c^{m-1}\bar{A}d^{m-1} \Rightarrow$
$a^{m-1}Ab^{m-1}a^n b^n c^n d^n c^{m-1}\bar{A}d^{m-1} \Rightarrow a^m b^m a^n b^n c^n d^n c^m d^m$ (see Figure 7).

$$a^{m-1}\,\underbrace{Ab^{m-1}a^{n-1}\,\underbrace{Ab^{n-1}c^{n-1}\bar{A}}\,d^{n-1}c^{m-1}\bar{A}}\,d^{m-1}$$

Fig. 7. For Example 4, the synchronization is by the paired brackets: the nonterminals are synchronized if the substring between them is in the bracketed language $D_{N,T}$.

Theorem 1. $L(BSRS) \subseteq L(SSRS_2)$.

Proof sketch. Given a *BSR* $G = (N_B, T, P, S)$ we construct a *SSR* $G' = (N', T, P', S')$, where $P' = \{(A_i \to c(x_1), \bar{A}_i \to c(x_2)) \mid (A \to x_1, \bar{A} \to x_2)\}$. □

Observation: An N, T-tree can be represented as a string by the mapping

$$str \ : \ \mathcal{T}_N(T) \to (N_B \cup T)^*, \text{ defined as}$$
$$str(t_a()) = a, \ a \in T \cup \{\lambda\} \text{ (a single-node tree labeled } a),$$
$$str(t_A(t_1, \cdots, t_m)) = A\, str(t_1) \cdots str(t_m)\, \bar{A}, \ A \in N$$
$$\text{(a tree labeled } A \text{ with subtrees } t_1, \cdots, t_m).$$

Likewise, words $w \in R_{N,T}$ can be mapped into N, T-trees by the mapping $tree : R_{N,T} \to \mathcal{T}_N(T)$

$$
\begin{aligned}
tree(a) &= t_a(),\ a \in T,\\
tree(A\, w_1 \cdots w_m\, \bar{A}) &= t_A(tree(w_1), \cdots, tree(w_m)),\ A \in N,\\
&\quad w_1, \cdots, w_m \in R_{N,T},\ m \geq 0,\\
R_{N,T} &= T \cup \bigcup_{A \in N} AD_{N_T}\bar{A}\\
(R_{N,T} &= \text{“rooted” bracketed language}).
\end{aligned}
$$

Observation: We recall that the foot of any auxiliary tree has an NA attribute. Without restricting the generality ((yield) language family), we also assume that the head labeled B of any auxiliary tree has an NA attribute (otherwise replace such a tree t by the tree $t_B^{NA}(t)$).

Theorem 2. $L(TAG) = L(BSRS)$.

Proof sketch.
$\subseteq$ For a TAG $G = (T, N, I, A, S)$ we construct the equivalent BSR system $G' = (T, N_B, P, S')$ with $S' = \{h(str(t)) \mid t \in I\}$ and $P = \{(B \to h(x_1), \bar{B} \to h(x_2)) \mid t = t_B(t_1, \cdots, t_m) \in A : str(t) = B^{NA}x_1B_*^{NA}x_2B^{\bar{N}A}\}$
$\supseteq$ For a BSR $G' = (T, N_B, P, S')$ we construct the equivalent TAG

$$
\begin{aligned}
&G = (T, N, I, A, S) \text{ where } S \text{ is a new symbol,}\\
&I = \{tree(S^{NA}w\bar{S}^{NA} \mid w \in S')\},\\
&A = \{tree(B^{NA}h^o(x_1)B_*^{NA}h^o(x_2)\bar{B}^{NA}) \mid (B \to x_1, \bar{B} \to x_2) \in P\},\\
&\quad h^o(B) = B^{OA},\ B \in N,\ h^o(a) = a,\ a \in T.
\end{aligned}
$$

Every derivation in G' corresponds to a derivation in G and vice-versa, so $L(G') = L(G)$. □

The notion of BSR can be extended, considering tuples instead of pairs of bracketed symbols, using the extended language $D_{N,T}^k$ (see Figure 8), defined by the grammar below

$$
\begin{aligned}
G^k &= (S, N_B^k \cup T, P, S),\ N_B^k = \{A^{(i)} \mid A \in N,\ 1 \leq i \leq k\},\\
P &= \{S \to \lambda, S \to SS\} \cup \{S \to a \mid a \in T\} \cup\\
&\quad \{S \to SA^{(1)}S \cdots SA^{(k)}S \mid A \in N\}.
\end{aligned}
$$

We can define $N_B^{\leq k}$ where the bracketed relation has at most k items. We define $BSRS_k$ as the set of all bracket-synchronized rewriting systems $G = (N_B^{\leq k}, T, P, S)$. We write $BSRS_* = \bigcup_{k \geq 1} BSRS_k$.
Observation: $\{a_1^n \cdots a_{2k}^n \mid n \geq 1\} \in L(BSRS_k)$.

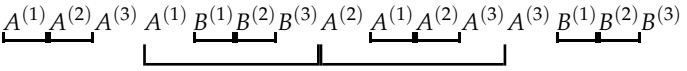

Fig. 8. For the extended BSR systems we show an example of the ternary synchronization.

5 Possible extensions and further research

After extending the BSR formalism we get a formalism that is very similar to Coupled Context-Free Grammars, or Klammergrammatiken [16]. We also find the step synchronized rewriting proprety defined as locality by Owen and Satta in [14]. Using Recursive Matrix Systems (RMS) [10] we can also simulate the derivations in SSRS.

Future research could go in several directions. First we could try to find a connection between EAs for TAG parsing and DNA computing.

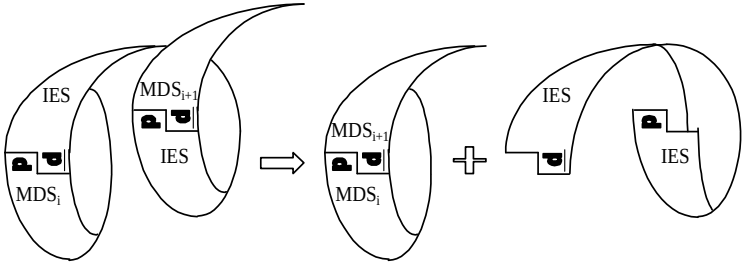

Fig. 9. LoopDirect (ld) operation in gene assembly in Ciliates

Both domains, EAs and DNA computing, need to code solution somehow for a given problem.

If we look at the process of gene assembly in Ciliates (we can find several Formal Frameworks described in [9]) we observe that parts of the DNA act as pointers as we can see in Figure 9 where MDS means Macronuclear Destined Sequences and IES is Internally Eliminated Sequences.

We believe that there could be a connection between the non-Terminals in BSR and non-overlapping pointers in self assembling in Ciliates and if we could do a synchronized instead of the elimination operation insertion as in Figure 10 we would obtain the adjoin operation as in Figure 5.

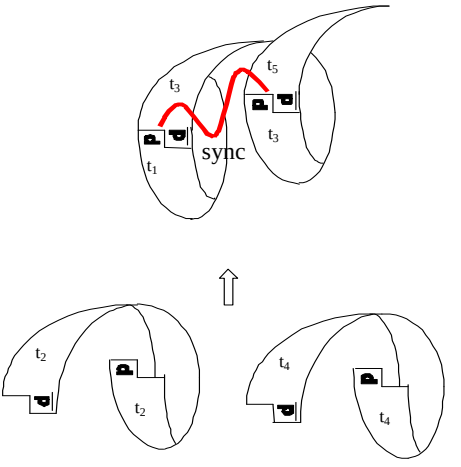

Fig. 10. Synchronized insertion in DNA molecules

As a second research direction we could try to simulate the BSR systems without rewriting rules, only adding contexts using Contextual Grammars [15].

Also, for future developments, we will run the EAs algorithms for more complex grammars including natural language

Parsing using the English Grammar available in the XTAG Project [19]. We will also try to guide the searching process of the EAs by 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. We will thus be able to use the result of the algorithm that 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.

6 Concluding remarks

We have proposed an Evolutionary Algorithm for Tree Adjoining Grammar parsing. We can observe that the classical parsing algorithm needs approximately 3 times more computations than the EA to solve the same problem.

One of the disadvantages of the EA parsing algorithm is that, for negative examples, the EA will not be able to say that there is no solution. We may not have let the algorithm run enough generations, but we ran some tests for positive examples and we approximated the requested number of generations required to find a solution for a certain length of the input string.

Finally, one interesting aspect of the EA parsing algorithm is that if the grammar is ambiguous, in one run of the same input string we found different parsings for the 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 for developing new models for knowledge-based representation systems, automatic text summarization etc.

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

References

1. Th. Bäck, *Evolutionary Algorithms in Theory and Practice - Evolution Strategies, Evoutionary Programming, Genetic Algorithms*, Oxford University Press, 1996
2. D. Beasley, D.R. Bull, R.R. Martin, *An Overview of Genetic Algorithms, Part 1*, Fundamentals, University Computing, 1993, 15(2) pp. 58-69.
3. D. Beasley, D.R. Bull, R.R. Martin, *An Overview of Genetic Algorithms, Part 2*, Research topics, University Computing, 1993, 15(4) pp. 170-181.
4. J. Earley, *An Efficient Context - Free Parsing Algorithm*, Communication ACM 13(2): 94 - 102, 1969
5. J.E. Hopcroft, J.D. Ullmann, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979
6. F. Gecseg, M. Steinby, Tree languages, *Handbook of Formal Languages* v. 3 (G. Rozenberg, A. Salomaa, eds.), Springer-Verlag, 1997, 1–68.
7. D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
8. D.E. Goldberg, *The Theory of Virtual Alphabets*, H.-P. Schwefel and R. Mnner, editors, Parallel problem Solving from Nature, pp. 13-22. Springer-Verlag, 1990.

9. T. Harju, I. Petre, G. Rozenberg, *Gene Assembly in Ciliates: Formal Frameworks*, Turku Centre for Computer Science, TUCS Technical Reports, NO 558, October 2003, ISBN 952-12-1233-0
10. D. Heckmann, *Recursive Matrix Systems Diplomarbeit* von Dominik Heckmann, angefertigt am Deutschen Forschungszentrum fr Knstliche Intelligenz GmbH und am Fachbereich 14, Informatik, Universitt des Saarlandes, Lehrstuhl Prof. Dr. Dr. h.c. Wolfgang Wahlster Saarbrcken 1999
11. A.K. Joshi, Y. Schabes, *Tree-Adjoining Garmmars*, G. Rozenberg, A. Salomaa, editors, *Handbook of Formal Languages*, Springer-Verlag, vol. 3, pp. 69-123, 1997
12. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer Verlag, 1992.
13. M. O'Neill, C. Ryan *Automatic Generation of Programs with Grammatical Evolution*, In Proceedings of AICS , pages 72-78, 1999
14. R. Owen, G. Satta, *Independent Parallelism in Finite Copying Parallel Rewriting Systems*, Theoretical Computer Science 223(1-2), 1999
15. G. Paun, *Marcus Contextual Grammars* Series : Studies in Linguistics and Philosophy , Vol. 67, 1998
16. G. Pitsch, *LR(k)-Coupled-Context-Free Grammars*, Inf. Process. Letter, 55(6): 349-358, 1995
17. Y. Schabes, A.K. Joshi *An Earley-Type Parser For Tree Adjoining Grammars*, Department of Computer and Information Science, University of Pennsylvania, 1988
18. P.H. Winston, *Artificial Intelligence*, third edition, Addison-Wesley, June 1992
19. The XTAG Project, http://www.cis.upenn.edu/∼xtag/