A New Evolutionary Parsing Algorithm for LTAG

Vijay Krishna Menon and K.P. Soman

Abstract Tree adjoining grammars (TAGs) are mildly context-sensitive psycholinguistic formalisms that are hard to parse. All standard TAG parsers have a worst-case complexity of O(n^6), despite being one of the most linguistically relevant grammars. For comprehensive syntax analysis, especially of ambiguous natural language constructs, most TAG parsers will have to run exhaustively, bringing them close to worst-case runtimes, in order to derive all possible parse trees. In this paper, we present a new and intuitive genetic algorithm, a few fitness functions and an implementation strategy for lexicalised-TAG parsing, so that we might get multiple ambiguous derivations efficiently.

Keywords Tree adjoining grammar · Evolutionary parsing · Genetic algorithm · Genetic operators · NLP · Syntax analysis · Derivation · Parse tree · Lexicalisation · Crossover · Mutation

1 Introduction

Tree adjoining grammars (TAGs) were proposed by Joshi et al. [1], to be used for natural language representation and processing. The grammar is a non-Chomskian formalism which is mildly context sensitive in nature. Unlike string generation grammars, TAGs use trees to be their elementary constructs. The benefit of using TAGs over generally popular grammars, such as context-free grammars (CFGs), is

V.K. Menon (✉)
Amrita School of Engineering, Center for Computational Engineering & Networking (CEN), Coimbatore, Tamil Nadu, India
e-mail: m_vijaykrishna@cb.amrita.edu

K.P. Soman
Amrita Vishwa Vidyapeetham, Amrita University, Coimbatore, Tamil Nadu, India
e-mail: kp_soman@amrita.edu

P.K. Sa et al. (eds.), *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*, Advances in Intelligent Systems and Computing 518,
DOI 10.1007/978-981-10-3373-5_45

that TAGs are able to capture a lot more linguistically and lexically relevant features which are normally lost in plain CFG models. TAGs have an *extended domain of locality* which captures furthest dependencies in a single rule. Furthermore, they are able to factor domain dependencies into sub-rules without affecting the parent's template. This gives TAGs an edge over other formalisms; one can model a language using fewer rules and capture its semantics (at least in a limited way) without a separate dependency parsing [2]. The TAG derivation is in fact a good dependency structure that we can use instead. In cases where a probabilistic parse is done, TAGs can almost compete with CFGs and the additional dependency parse required complementing the syntax trees.

The problem we face with TAGs is its exponentially worse parsing complexity for longer sentences; with multiple parse trees (ambiguous), this would be an exhaustive problem. We want all ambiguous parses of a given phrase or sentence. This will push the parser to worst-case scenarios. This was the main motivation to consider alternates that fish out multiple solutions (or optimums in certain cases); genetic algorithm seemed a good candidate.

TAG G is defined as a quintuple in [3] as follows:

$$\mathrm{G_{TAG}} = (\mathrm{N}, \mathrm{L}, \mathrm{T_I}, \mathrm{T_A}, \mathrm{S}) \tag{1}$$

where N is the set of all non-terminals, L is the set of all terminals, T_I is the set of all *initial trees,* T_A is the set of *auxiliary trees* and S is a sentential start symbol. Trees that have a root node named S are called *sentential trees.* TAGs generate trees and not strings. The trees conjoin, using two operations, namely *adjunction* and *substitution*. Substitution is a nominal operation where two initial trees merge at a node that is marked for substitution. This is the same substitution that results in the middle-out growth of a CFG sentential form; essentially, it is a CFG operation. One tree is a parent and the other is the child. The parent's node which is marked for substitution is replaced by the child's root node, *attaching* the entire child tree with the parent. This is essentially possible *iff* the substitution node is a leaf (external) node.

Adjunctions on the other hand are *inserting* auxiliary trees to initial trees. An auxiliary tree has got a root node and an identical *foot node*. The concept of adjunction is splitting a node of the parent tree horizontally into a root and foot nodes of an auxiliary tree. The criterion is the same as substitution except that it can be done on any node (mostly internal nodes). While the root of the auxiliary tree replaces the adjunction node in the parent, the foot node of the auxiliary tree will adopt the sub-tree of the same node being replaced, so essentially inserting the auxiliary tree into the initial tree.

Adjunctions make TAG mildly context sensitive. Figure 1 illustrates how both operations on the tree forms eventually affect the string yield from the final derived tree. Figure 2 pictures the physical process of these operations, a simple attachment for substitutions and a partly complex insertion for adjunctions. For more details on TAGs, refer works of Aravind Joshi, Vijay-Shanker, Yves Schabes and M Shieber [3–5].

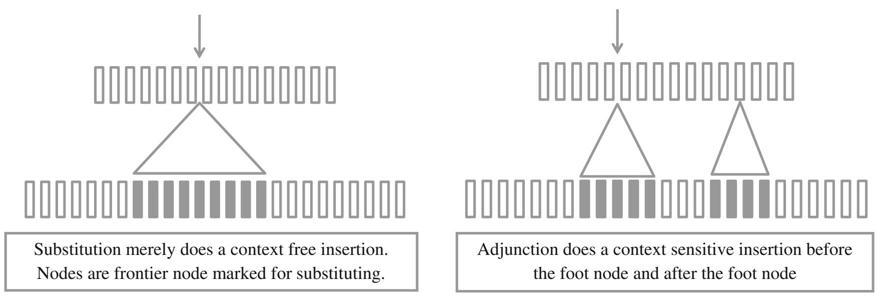
Substitution merely does a context free insertion.
Nodes are frontier node marked for substituting.
Adjunction does a context sensitive insertion before the foot node and after the foot node

Fig. 1 Yield of substitution operations between two initial trees (*left*) and yield of adjunction operation between an initial and an auxiliary tree (*right*). Clearly, adjunction does a context-sensitive insertion preserving the foot node and its yield intact while insert before and after it

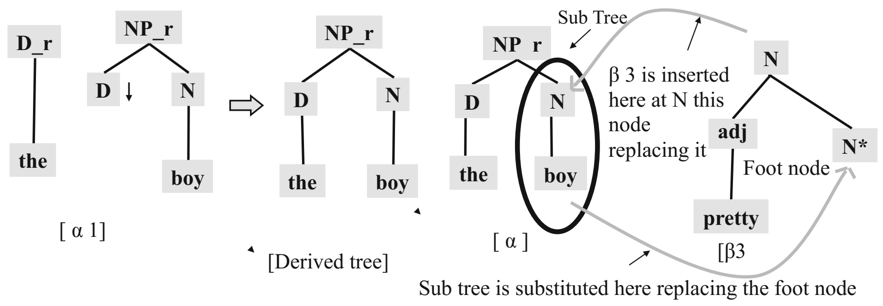
D_r
the
NP_r
D ↓
N
boy
[α 1]
NP_r
D
N
the
boy
[Derived tree]
Sub Tree
NP r
D
N
the
boy
[α]
β 3 is inserted here at N this node replacing it
N
adj
N*
Foot node
pretty
[β3
Sub tree is substituted here replacing the foot node

Fig. 2 Substitution is a single attachment, while adjunction does two separate attachments. The classical parsing treats adjunction as partial jump and completion process requiring a *left* completion and a *right* prediction before the insertion is completed

2 Prior Works on Evolutionary Parsing

There are many classical and conventional algorithms to parse TAGs. The popular algorithms are detailed in [4], a CYK-type parser, and [6], an Earley-type parser. Our comparing implementation is the later with an obvious difference that it is a multithreaded parser rather than the backtracking version detailed in [3, 6]. For more details on this please refer our prior published works [7, 8].

The main work of relevance focused on combining EA and TAG parsing is $EATAG_P$ [9], where an evolutionary algorithm has been proposed to parse TAGs which they demonstrate on a simple copy language. The implementations are done on different platforms so they have given a comparison of the number of computations required in each case. Their results show $EATAG_P$ to be much more efficient not just for one parse but also asymptotically too as the EA is able to fit a linear order while the classical TAG parsing will have an exponential order.

The EA version is a randomised search so evidently it gives different values as for the population size and for the selection in each generation; computations vary for different executions of the parser with the same string. To finalise the number of computations, it is averaged across minimum 10 runs, for comparison. The main focus of the paper is to find the right derived (parse) trees using *gene pools* created from (tree, node) pairs that will be progressively added to a chromosome eventually tracing out the *derived tree* with the desired yield. The fitness function is rather vaguely defined, but the strategy is clear. They use multiple fitness scores with decreasing priority like a three-tuple score vector *(matches, coverage and yield)*. The matches record the continuous matches of words in the input string order, the coverage is the total word matches, and the yield tracks the length overshoot of the chromosome over the real string with a negative value. They claim to have used cubic order fitness first and successively reduced the order eventually using a linear function which gave best results. It also merits mentioning some other basic works on evolutionary parsing such as the genetic algorithm-based parser for stochastic CFGs detailed in [10]. This work describes a lot of vague areas when it comes to GA-based parsing. The grammar is considered to be probabilistic giving an easy way to evaluate the sentential forms, in order to rank and compare different individual parses.

The concept of *coherent genes* is introduced in [10] and is a concept that is absent in the $EATAG_P$ where they have eliminated the non-viable gene issue by carefully biasing the initial gene pool and using it in a circular manner. In CFG, however, the sentential forms are string and this will not be a problem. In fact, this gives a better fitness criterion to validate and evaluate individuals in a population based on relative coherence. However, this approach works mostly on non-lexicalised grammars by grouping the words based on POS categories.

3 Genetic Algorithm (GA) Model and Operators

Our approach to evolutionary parsing is to generate random derivations with pre-assembled genes. In TAG parsing, derived trees or parse trees give only syntax information, while other relevant linguistic attributes, such as dependency, semantics and lexical features, are all lost. Undoubtedly, the more useful parse output is not the parse tree but the know-how of creating one. In TAG terminology, we call this the derivation structure, which is normally represented as a tree. It gives obvious advantages to fetch derivations rather than just parse trees.

We rank derivations of an input string initially generated randomly and pruned using the GA process to some threshold fitness value. The random derivations are indicative of individual parses in our algorithm. To represent various genes is the challenge as the derivation nodes contain a lot of information. Thus, we defined TAG parsing as an eight-tuple GA process.

$$\mathrm{GATAG_P} = \{\mathrm{W_D}, \Gamma_\mathrm{I}, \Gamma_\mathrm{C}, \mathrm{f_{cover}}, \mathrm{f_{cohere}}, \Omega_\mathrm{LR}, \mathrm{P_{derivation}}, \mathrm{C_{stop}}\} \tag{2}$$

where

- W_D is a complete lexical dictionary of words. The lexicalisation can be treated as an indexing of $\mathrm{W_D}$. These aspects are discussed in detail in a later section.
- Γ_I is set of genes that start a parse (sentential genes).
- Γ_C is the set of all other genes that relates to each lexical item.
- f_{cohere} is a fitness score that measure total coherence between genes in a chromosome.
- f_{cover} is the coverage of lexicons by various chromosomes.
- Ω_{LR} is a genetic operator that only yields one child (the left-right child).
- $P_{\mathrm{derivation}}$ is initially a random population of chromosomal individuals.
- C_{stop} is a termination criterion for the GA, also called the stopping condition. This is usually a preposition that needs to be realised for the GA process to stop.

The *stop condition* for the GA process is that average fitness be greater than a threshold value. This value, however, needs to be estimated based on empirical observation of multiple runs on the process itself. There are some other such parameters that too require similar estimations. To understand more of these parameters and their statistical properties, we can define a convex problem for it that gives better mathematical grounds for analysing them. This, however, can be tabled for another publication as it is not in focus here.

3.1 Genes, Gene Pools and Coherency

For a complete representation of a derivation tree in TAG, we need information as to the main tree that will be lexicalised with the matrix verb of the sentence. These are sentential constructs which are also initial type trees; we call them *sentence-initial* trees. In order for us to initiate a parse on a sentence, we need such a tree. In an ambiguous grammar, there can be multiple trees which can initiate a parse on the sentence. So this has to be forced into the algorithm that the first tree it selects will be a sentence-initial tree. Hence, genes which shall code for these trees are unique and needs to be handled separately.

For representing a gene in any derivation string, we need the following two-tuple and four-tuple structures.

$$\gamma_\mathrm{I} = \{\mathrm{t_I}, \mathrm{l_{W_D}}\}, \gamma_\mathrm{I} \in \Gamma_\mathrm{I}, \gamma_\mathrm{I} \in \Gamma_\mathrm{I}, \mathrm{T_I^S} \subset \mathrm{T_I}, \mathrm{l_{W_D}} \in \mathrm{W_D} \tag{3}$$

$$\begin{gathered}\gamma_\mathrm{C} = \{\mathrm{t_N}, \mathrm{l_{W_D}}, \mathrm{t_p}, \mathrm{n_{t_p}}\} \\ \gamma_\mathrm{C} \in \Gamma_\mathrm{C}, \mathrm{t_N} \in (\mathrm{T_I} \cup \mathrm{T_A} - \mathrm{T_I^S}), \mathrm{T_I^S} \subset \mathrm{T_I}, \mathrm{l_{W_D}} \in \mathrm{W_D}, \mathrm{t_p} \in (\mathrm{T_I} \cup \mathrm{T_A})\end{gathered} \tag{4}$$

The genes are of two types the sentence-initial (SI) genes in (3) and the common genes in (4). The sets are generated into separate gene pools. The SI-type gene γ_I is simple enough; they encode the root node of derivation with a sentence-initial tree and a possible lexicalisation with any occurring word l_W. The common gene is a bit more complicated. This encodes all the other nodes of the derivation and has two extra items: the parent tree t_p and the node of the parent tree n_p, where this given tree t_N has attached itself and lexicalised with l_W.

The genes in true nature represent all possible nodes in all derivation trees. This is the reason we must try and fit the nodes in the right tree. The creation of these is a batch process that is $O(w.k^2)$ in time where there are k trees and w words. This can also be done in a single initialisation process and not repeat it for every parse again and again. We have evaluated our algorithm purely based on the fitness scoring and selection process and have omitted gene creation from it, in order to get an idea of the efficiency as a function of the length of the sentence, and compare it to classical TAG parsing.

Coherency in a gene is defined as the viability of the gene to exist in a real derivation. This is a binary property, and the gene can be non-coherent in many ways: if the tree is not lexicalised by the paired lexicon, if the parent tree does not have a parent node specified, or if the main tree can never conjoin with the specified parent site due to adjunction constraints or lack of substitution marker. These can be verified while the gene is being created as a set of check heuristics, thus never creating a non-viable gene in the gene pools. The benefit of this process is we can bias it with more heuristics if required. This justifies our call to make this a batch process for the bulk of trees and lexicons, as once the coherent genes are created, then it is just a matter of indexing them to create local gene pool for each sentence.

3.2 Chromosomes and Fitness

A chromosome is a complete derivation tree. It contains genes that code for every word of the given sentence. While the coherence of genes can be ensured, chromosomes are randomised and may be non-coherent. This can be refined only through genetic evolution (GA) process. Since the gene that starts the parse needs to be added separately, the chromosomes are built by this bias. A chromosome is defined as an ordered set of genes as follows

$$\kappa_i = \{\gamma_I \prod_{j=1}^{w-1} \gamma_{C_j} : \gamma_I \in \Gamma_I \text{ and } \forall j\ \gamma_{C_j} \in \Gamma_C\}, \text{w is length of the sentence} \tag{5}$$

Note that it is concatenation of genes and the order of genes is the same as the order of the lexicons (lexicons) in the sentence. We are not doing any other kind of biasing for continuous genes in the above formation. We have empirically found that a population of over twenty thousand per instance is a good pool to comprehensively collect most syntactic ambiguities.

Coherence over chromosomes is defined as a fitness score unlike in genes. The max score is $|w\text{-}1|$, and min score is 0. It is a count of the number of cohering genes; if a gene with parent t_p exists, then there should be a gene with tree t_p in the same chromosome. If a single non-coherent gene exists, the derivation becomes non-viable. However, coherence can be improved by genetic evolution, by applying genetic operators.

The fitness calculation is done using a search and a matching heuristic that iterates over the entire chromosome to find gene pairs that match. It is possible to do it exhaustively using an $O(n.\log_2(n))$ pair comparison like a sort process. This can be replaced by a shallow parse that will take an $O(n^3)$ complexity. However, it can also be achieved with a linear heuristic function. Out of all these, the shallow parse version is the most accurate and gives almost all the ambiguous derivations. The other techniques yield good results faster but also some false positives that need to be additionally filtered.

Coherence is secondary here, and there is a more important fitness concern that we need to solve, the *word coverage* problem. To make sure that the chromosome has complete word coverage, that is, to check that there is a gene in the chromosome for every word in the given sentence, we employ a simple linear search logic that marks words as each gene is iterated on. The coverage problem is given due importance in the earlier generations, and later on, the coherence will take over while coverage will still be active at a minimal level.

3.3 The Left-Right Genetic Operations

The main biasing of our GA model is that there are fewer random operators than most classic GAs. The operator we have used is a crossover and biased mutation for the main selection process. Our crossover genetic operator works by finding exactly one child per pair of parents. This is the leading child or as to call it, the LR child. Since our parse and coherency model works from left to right, we get better and faster convergence from it. The operator can be defined as follows

$$\Omega_{C_{LR}}(\kappa_i) = \kappa_i^{LR} \text{ such that } \kappa_i^{LR} = \{\gamma_I \prod_{j=1}^{h} \gamma_{C_j} \prod_{k=h+1}^{w} \gamma_{C_k} : \forall j\ \gamma_{C_j} \in \kappa_i \text{ and } \forall k\, \gamma_{C_k} \in \kappa_{Random}\},\ h \text{ is the coherence index of } \kappa_i \tag{6}$$

As it follows, the crossover needs a good husband and randomly selects a wife to crossover, preserving the husband's coherence and fitness. The idea is never to create a less fit offspring from any husband. So naturally the evolution will go forward. Similarly, biased mutation is also defined. The operator never lowers the fitness of a chromosome it is operating on. The mutation can be defined as follows.

$$\begin{aligned}&\Omega_{M_{LR}}(\kappa_i) = \kappa_i^M \text{such that}\\ &\kappa_i^M = \{\gamma_I \prod_{j=1}^{h} \gamma_{C_j} \prod_{k=h+1}^{m-1} \gamma_{C_k}\gamma_{C_{Random}} \prod_{k=m+1}^{w} \gamma_{C_k} : \forall j\ \gamma_{C_j} \in \kappa_i \text{ and}\\ &\forall k\ \gamma_{C_k} \in \kappa_i,\ \gamma_{C_{Random}} \in \Gamma_C\}, \text{h is the coherence index of } \kappa_i\\ &\text{and m is the mutation index such that } h < m < w\end{aligned} \tag{7}$$

4 Implementation

The primary implementation is done using a pruned XTAG subset for English [8]. The main assumption as we have discussed earlier, maintains that the TAG should be single anchored and is lexicalised with only one word. This is why we associate a tree with just one lexicon in our gene model. We have used the O(n.log(n)) fitness strategy which we introduce earlier.

The *howCover* method for estimating word coverage works by directly counting the genes that code for each word. The total count must be equal to the count of the words in the input sentence. This algorithm works linearly by using a word-hashed search of the required gene pool.

```
method howCover($k_i$, w)
    cover <- 0
    for j from 0 until |w|
        if $k_i$ contains $w_j$ then increment cover
    rof
    return cover
end
```

The *howCohere* method reads continuous coherence and returns when any gene in the sequence is non-cohesive. Since it employs a *binary search* on genes, the chromosome is required to be sorted on parent-node ordering. This is an additional overhead for the sake of lessoning this computation. In real practice, we can avoid this sorting too. This will be demanding as the length of the string increases; we would need a real big population size to make sure enough viable chromosomes exist. The above method is of *O(n.log(n))*.

```
method howCohere ( $k_i$ )
    cohere <- 0
    for $g_j \in k_i$, j = 1, 2 … $|k_i|$
        if bSearch ($k_i$, $g_j.t_p$, $g_j.n_p$)
            increment cohere
        else return *cohere*
        fi
    rof
end
```

The selection is yet again a straight process. Once the genetic operations are defined, then the selector simply calls them to incrementally evolve and transform the population. Typically within 10–15 generation, we observed convergence. The challenge the selector faces is how to eliminate duplicate chromosomes and duplicate solutions. The first way is to make the population into a set so duplicates will never be stored. The second way is to eliminate duplicates when they are created during the process, by making the solution holder a binary tree or a set. One observation we have made is that the duplication of initial chromosomes is less as it is created in a sequentially random process (no re-seeding of the random generator). So we have used a binary tree to check for duplicates in the solution holder. We can extend our model for a wide coverage grammar like XTAG [11], but requires some overhaul of our base GA definition and will incur considerable complexity for coherency computations.

```
method selector ( $P_D$, $sols_{set}$, $good_{set}$ )
    for $k_i \in P_D$, i = 0, 1, 2 … $|P_D|$
        if howCover( $k_i$ ) and howCohere( $k_i$ ) are maximum
            add $k_i$ to $sols_{set}$
        else if only howCover($k_i$)is maximum
            add $k_i$ to $good_{set}$
        else add mutateLR($k_i$) to $newP_D$
        fi
    rof
    if $good_{set}$ is not empty
        $newP_D$ <- crossoverLR( good , $P_D$ )
        else $newP_D$ <- crossoverLR( Random($P_D$) , $P_D$)
    fi
    return $newP_D$
end
```

5 Results and Conclusion

The proposed algorithm theoretically fares best when we have a word size of seven or more words. The overall complexity of $GATAG_P$ is $O(n.log(n).|P_D|)$. The implementation front we incur some extra overheads and the efficient size of the input sentence is observed to be eight. Our main obstacle to this comparison is the false positives that we incur which needs to be filtered using shallow parsing. We have presented in Table 1, the theoretical speed up between classical and evolutionary TAG parsing. For this analysis, we assume the population size to be 10000 with ten generations. We have also empirically observed that the number of trees selected per sentence is on an average 41. The worst-case complexity for

Table 1 A theoretical computations chart for GATAG$_P$ and Earley-type parser for different input lengths. The calculations assume a population of 10000 chromosomes and a tree pool of 41 trees on an average. The complexity of former is $O(n.log_2(n).|P_D|)$ and latter is $O(|G|.n^6)$

|w|	GATAG$_P$	Earley type	|w|	GATAG$_P$	Earley type
7	2665148	4823609	12	5501955	122425344
8	3200000	10747904	13	6110572	197899169
9	3752933	21789081	14	6730297	308710976
11	4905375	72634001	15	7360336	467015625

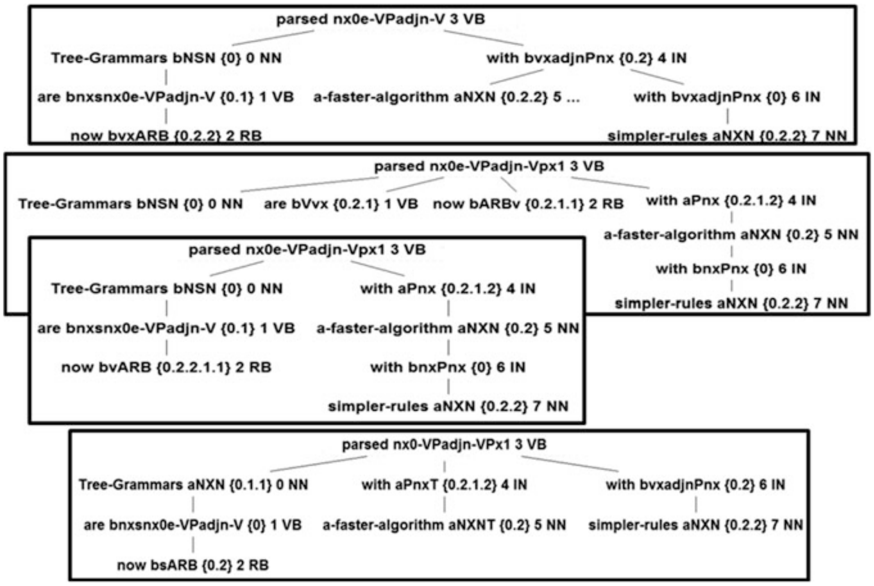
parsed nx0e-VPadjn-V 3 VB
Tree-Grammars bNSN {0} 0 NN
with bvxadjnPnx {0.2} 4 IN
are bnxsnx0e-VPadjn-V {0.1} 1 VB
a-faster-algorithm aNXN {0.2.2} 5 ...
with bvxadjnPnx {0} 6 IN
now bvxARB {0.2.2} 2 RB
simpler-rules aNXN {0.2.2} 7 NN
parsed nx0e-VPadjn-Vpx1 3 VB
Tree-Grammars bNSN {0} 0 NN
are bVvx {0.2.1} 1 VB
now bARBv {0.2.1.1} 2 RB
with aPnx {0.2.1.2} 4 IN
a-faster-algorithm aNXN {0.2} 5 NN
with bnxPnx {0} 6 IN
simpler-rules aNXN {0.2.2} 7 NN
parsed nx0e-VPadjn-Vpx1 3 VB
Tree-Grammars bNSN {0} 0 NN
with aPnx {0.2.1.2} 4 IN
are bnxsnx0e-VPadjn-V {0.1} 1 VB
a-faster-algorithm aNXN {0.2} 5 NN
now bvARB {0.2.2.1.1} 2 RB
with bnxPnx {0} 6 IN
simpler-rules aNXN {0.2.2} 7 NN
parsed nx0-VPadjn-VPx1 3 VB
Tree-Grammars aNXN {0.1.1} 0 NN
with aPnxT {0.2.1.2} 4 IN
with bvxadjnPnx {0.2} 6 IN
are bnxsnx0e-VPadjn-V {0} 1 VB
a-faster-algorithm aNXNT {0.2} 5 NN
simpler-rules aNXN {0.2.2} 7 NN
now bsARB {0.2} 2 RB

Fig. 3 Some derivations created by parsing the example sentence 'Tree Grammars are now parsed with a faster algorithm with simpler rules'. The sentence yielded 65 derivations and parsed in less than 10 s. The same sentence gave 40 derivations in the classical Earley-Type parser and took 25 s

Earley-type TAG parser is $O(|G|^2.n^9)$ [6], but ours being a multithreaded parser we can assume the minimum worse case (time) for TAG parsing as $O(|G|.n^6)$ [7]. Figure 3 gives screen shots of a few syntactically ambiguous derivations given by the GATAG$_P$ for the example sentence *'Tree Grammars are now parsed with a faster algorithm with simpler rules'*. The trees are generated from solution set chromosomes, using our *TAG Genie* [7] tree viewer.

References

1. Joshi, A. K., Levy, L. S., Takahashi, M.: Tree adjunct grammars. Journal of Computer and System Sciences. 10, 1, 136–163 (1975).
2. Schuler, W.: Preserving semantic dependencies in synchronous Tree Adjoining Grammar. In: Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics (ACL'99). Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 88–95 (1999).
3. Joshi, A. K., Schabes, Y.: Tree-adjoining grammars. Handbook of formal languages, vol. 3, Rozenberg, G. and Salomaa, A., (eds.). Springer-Verlag New York, Inc., USA, pp. 69–123. (1997).
4. Vijay-Shankar, K., Joshi, A. K.: Some computational properties of Tree Adjoining Grammars. In: Proceedings of the 23rd annual meeting on Association for Computational Linguistics (ACL'85). Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 82–93. (1985).
5. Shieber, S. M., Schabes, Y.: Synchronous tree-adjoining grammars. In: Proceedings of the 13th conference on Computational linguistics - Volume 3 (COLING'90), Karlgren, H. (Ed.). Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 253–258. (1990).
6. Schabes, Y., Joshi, A. K.: An Earley-type parsing algorithm for Tree Adjoining Grammars. In: Proceedings of the 26th annual meeting on Association for Computational Linguistics (ACL'88). Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 258–269 (1988).
7. Menon, V. K.: English to Indian Languages Machine Translation using LTAG. CEN, Amrita Vishwa Vidyapeetham University. Master's Thesis. Coimbatore, India, doi:10.13140/RG.2.1.5078.5048 (2008).
8. Menon, V. K., Rajendran, S., Soman, K. P.: A Synchronised Tree Adjoining Grammar for English to Tamil Machine Translation. In: International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, pp. 1497–1501, doi:10.1109/ICACCI.2015.7275824. (2015).
9. Dediu, A. H., Tîrnauca, C. I.: Parsing Tree Adjoining Grammars using Evolutionary Algorithms. In: ICAART. (2009).
10. Araujo, L.: Evolutionary Parsing for a Probabilistic Context Free Grammar. In: Revised Papers from the Second International Conference on Rough Sets and Current Trends in Computing (RSCTC '00). Ziarko, W. and Yao, Y. Y., (eds.). Springer-Verlag, London, UK, pp. 590–597. (2000).
11. The XTAG Research Group, University of Pennsylvania, http://www.cis.upenn.edu/~xtag/tech-report/.