PLTIG Parsing

Winston Cheng
cheng3@fas.harvard.edu

Paul Govereau
govereau@eecs.harvard.edu

Dan Mauer
mauer@fas.harvard.edu

Alexander Rush
rush@fas.harvard.edu

January 13, 2005

Abstract

We describe the TIG formalism and its advantages over CFGs and TAGs. We then give an algorithm for training PLTIGs following Hwa (2001). We train on a partially supervised corpus by repeated parsing with a lexicalized normal form. The training algorithm is EM-style, and the parsing algorithm is a generalization of CKY. Finally, we look toward the possibility of extending this method into a synchronous form.

1 Introduction

In this paper, we describe the implementation of a training and a parsing algorithm for probabilistic, natural-language grammars. Following Hwa (2001), we describe a training algorithm and a parsing algorithm for Probabilistic Lexicalized Tree-Insertion Grammars (PLTIG). TIGs are a simplified form of Tree-Adjunction Grammars (TAG) (Joshi, 1975; Joshi and Schabes, 1997).

TAGs have proven to be useful for describing linguistic properties; however, they are computationally expensive. The parsing time for a TAG is $O(n^6)$. TIGs are more efficient with a time complexity of only $O(n^3)$.

While more efficient, TIGs are also less expressive than TAGs. TAGs can recognize some context-sensitive languages, and are therefore more expressive than Context-Free Grammars (CFGs). It has been shown that TIGs are strongly equivalent to normal Context-Free Grammars (Schabes and Waters, 1995). That is, every TIG can be converted into a weakly equivalent CFG, and every CFG can be converted into a strongly equivalent TIG.

Even though TIGs are equivalent to CFGs, there are a number of reasons to prefer TIGs for describing natural languages.

1. TIGs are more natural for describing linguistic properties. For instance, representing relationships between words that may be relatively far apart in a sentence can be difficult with a CFG, whereas in a TIG this is quite easy.
2. TIGs are much more compact than their corresponding CFGs.
3. TIGs are easier to lexicalize (Schabes and Waters, 1995).
4. TIGs can be synchronized similar to Synchronous TAGs(Shieber and Schabes, 1990). This allows for the possibility of adapting the PLTIG algorithms for machine translation (Weaver, 1955; Brown et al., 1990).

The PLTIG training algorithm requires a training corpus. The algorithm described here can be run supervised or unsupervised. For our experiments, we used a hand-parsed corpus with part-of-speech tags. Details of the implementation can be found in Section 6.

In the next section, we will describe PLTIGs. We then go on to describe the training algorithm, a CKY

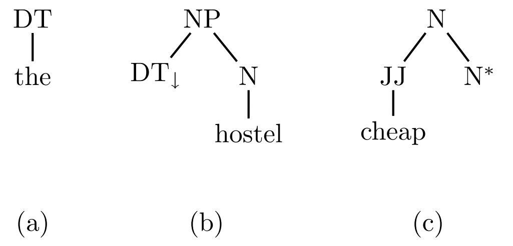


Figure 1: Examples of Elementary Trees. (a) A simple initial tree. (b) An initial tree with one substitution node. (c) An auxiliary tree with foot node N.

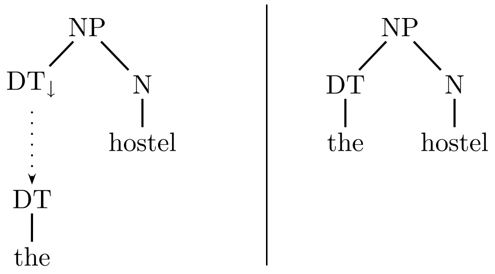


Figure 2: The substitution operation.

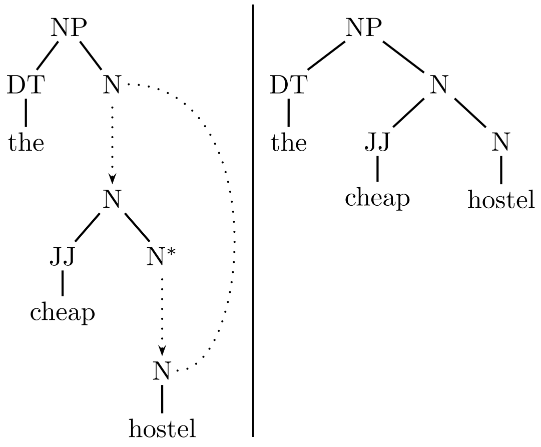


Figure 3: Adjunction

parsing algorithm and the calculation of probabilities. Finally, we describe some details of our implementation and conclude with some experimental results and directions for future work.

2 Tree-Insertion Grammars

Tree-Insertion Grammars (TIG) are a simplification of the more general framework of Tree-Adjunction Grammars (TAG), a form of tree-rewriting system (Joshi, 1975; Joshi and Schabes, 1997). The languages recognized by TIGs are a proper subset of the languages recognized by TAGs. We will start with a description of TAGs and then note how TIGs differ.

2.1 TAGs

A TAG consists of a set of *elementary trees* and operations for combining trees. There are two types of elementary trees: *initial trees* and *auxiliary trees.* Initial trees may contain nonterminal leaf nodes. The nonterminal leaf nodes may be marked as valid *substitution* nodes using the subscript "$_\downarrow$". Figure (1) shows two examples of initial trees. Tree (a) is a simple initial tree with no nonterminal leaf nodes. Tree (b) is an initial tree with one nonterminal leaf node, DT, which is a valid substitution node.

Auxiliary trees are similar to initial trees, but they also contain a special nonterminal leaf node with the same label as the root node. This node is referred to as the *foot node*, and is denoted with the superscript "*". Tree (c) in Figure (1) is an example of an auxiliary tree with foot node N.

There are two operations for combining trees, *substitution* and *adjunction.* Substitution allows us to replace a substitution node with another tree that has a root node with the same nonterminal symbol. For example, using the trees from Figure (1), we may replace the DT node in Tree (b) with Tree (a) to produce a new tree. Figure (2) illustrates the substitution operation described above.

Auxiliary trees can be combined with other trees using the adjunction operation. Adjunction allows an auxiliary tree to be inserted into another tree at an internal node. The root node of an auxiliary tree must match the internal node where it is inserted. The children of the internal node are moved to the foot node of the inserted auxiliary tree. Figure (3)

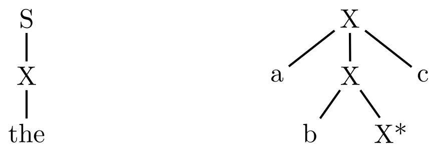


Figure 4: A TAG for the language $a^n b^n c^n$

demonstrates the adjunction of Tree (c) to the internal node N of the tree from Figure (2).

A TAG grammar can express languages that are *mildly context-sensitive*; that is, TAGs are strictly more expressive than Context-Free Grammars, but less expressive than context-sensitive grammars. It is easy to see that TAGs are more powerful than CFGs. Figure (4) shows a simple TAG that recognizes the language $a^n b^n c^n$ which cannot be recognized be a CFG[1]. The context-sensitivity in this example comes from the fact that the auxiliary tree's foot node has lexical material to both the left and right. Therefore, strings in the parse tree to the left and right of a foot node can be dependent as in our example. Auxiliary trees such as this are referred to as *wrapping trees.* The existence of wrapping trees complicates the TAG formalism to a surprising degree. The primary difference between TAGs and TIGs is that wrapping trees are not allowed in TIGs.

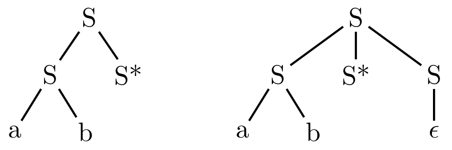
2.2 TIGs

A TIG is a TAG where wrapping auxiliary trees are disallowed. None of the elementary auxiliary-trees can be wrapping trees, and furthermore, no adjunction is allowed to create a wrapping tree. This restriction is enforced by:

- Requiring all elementary auxiliary-trees to be one of two possible types: *left-auxiliary* or *right-auxiliary.* Left- and right-auxiliary trees have all of their non-empty frontier nodes to the left or right, respectively, of the foot node.
- Disallowing adjunctions of auxiliary trees when this would result in a wrapping tree. An invalid adjunction can occur only when the two trees are of different types.

[1]This example comes from Hwa (2001)

Below are two examples of valid left-auxiliary trees.



Although the second tree is technically a valid left-auxiliary tree, we will avoid using such trees in the sequel.

2.3 PLTIGs

Probabilistic Lexicalized Tree-Insertion Grammars (PLTIG) are an extended TIG formulation. The two extensions added are *lexicalization* and *probabilities.* These two extensions are orthogonal, and we will describe each in turn.

Lexicalization A lexicalized TIG is a TIG in which all of the elementary trees contain at least one lexical (terminal) leaf node. By requiring that all trees contain a lexical leaf node we ensure that derivations represent both the parsing tree structure and the actual words. This is particularly important as probabilistic grammars are being trained on a corpus of text. The training data contains only words, so it is helpful if the elementary trees also contain these words for training purposes.

Probabilities A probabilistic TIG assigns a probability to each possible substitution or adjunction in a grammar. With these, we can compute the probabilities of different parses of a sentence. In order to perform probabilistic parsing, we need to add four sets of parameters to the elementary trees and their nonterminal nodes.

First, for each elementary tree t, there is a probability $p_i(t)$ of being the start tree for a derivation. For auxiliary trees this probability will be zero, and for initial trees the probabilities must be normalized such that

$$\sum_t p_i(t) = 1 \qquad .$$

Second, for each substitution node η, there is the probability for each initial tree t to be substituted at that node $p_s(\eta, t)$. These probabilities make up a distribution that must be normalized such that for each η

$$\sum_t p_s(\eta, t) = 1 \qquad .$$

Third, for each nonterminal, internal node η, there is a probability for each auxiliary tree t to adjoin to the left $p_l(\eta, t)$ or to the right $p_r(\eta, t)$. It is also possible that no auxiliary trees will adjoin to either the left $p_{nl}(\eta)$ or the right $p_{nr}(\eta)$. All of these parameters must be normalized such that for each η

$$p_{nl}(\eta) + \sum_t p_l(\eta, t) = 1$$

and

$$p_{nr}(\eta) + \sum_t p_r(\eta, t) = 1 \qquad .$$

Finally, for each nonterminal, internal node η, there may be a simultaneous left and right adjunction. In such a case, we have a probability that the right adjunction takes place first $p_{rl}(\eta)$ and vice-versa $p_{lr}(\eta)$. These parameters must also be normalized.

$$p_{rl}(\eta) + p_{lr}(\eta) = 1$$

The goal of our system is to automatically learn the best values for each of these parameters from a corpus of text. In the next section we describe the algorithm used for learning these parameters.

3 EM

In order to learn a Tree Insertion Grammar for building parses of English language sentences, we use a PLTIG induction method devised by Hwa (2001) which is based on an implementation of the general Expectation Maximization (EM) algorithm (AP Dempster and Rubin, 1977). We will now summarize the general algorithm and explain the specific implementation used to induce TIGs.

The EM algorithm is built around the concept of Maximum Likelihood Estimation: given a data sample, we must determine the parameters θ to plug into a given model that maximize the likelihood that the model will produce the examples on which it was trained - in our case, the model is a PLTIG, the parameters are the adjunction probabilities of that PLTIG, and the examples are correct, derived parses of English sentences.

If we had every possible valid parse of every possible English sentence (the *sufficient statistics* of the data), we could trivially compute the ideal parameters for the model. Since we do not have sample data approaching that level of completeness, we must instead estimate these parameters.

For the moment, assume that we will be able to learn the model parameters from the data we do have; this will allow us to compute a probability distribution over the unobserved data. In our case, for each adjunction probability, we can use our learned parameters to along with the observed data to estimate what we expect would be found in the remaining, unobserved data. This will give us an estimate of the sufficient statistics (i.e., the *expected sufficient statistics*) by calculating the expected probabilities of each adjunction action across all data, observed and missing.

EM is an iterative algorithm that makes use of these concepts as follows. It starts by setting θ to some initial values. Once the θ is set, EM uses it along with the observed sample data to compute the expected sufficient statistics of the complete (observed + missing) data. New values for θ are then chosen to maximize the likelihood of the expected sufficient statistics. The new θ is used to re-estimate the expected sufficient statistics, and the process goes back and forth until convergence, usually defined as the point at which the difference between two successive sets of θ is below some threshold.

A concise representation of the full algorithm can be seen in Algorithm 1. We use the following notation, adapted from that used by Hwa (2001). An observed training sentence is denoted $\mathbf{w} = w_1, ..., w_t$. Each possible derivation of $\mathbf{w}$ is represented as $\mathbf{v} \in \mathcal{V}$ where $\mathcal{V}$ is the set of all such derivations. Each derivation is comprised of several steps denoted $v_{(s,t)} \in \mathbf{v}$ where $v_{(s,t)}$ is the step which forms the constituent covering string positions $w_{s+1}, ..., w_t$ in the parse tree. Each derivation can also be represented as a series of adjunction operations: $\text{act}(v_{(s,t)}) = \alpha_{ij}$ means that the adjunction opearation α_{ij} – the adjunction of

```
Initialization step
Build initial grammar in Hwa normal form
for $\eta_i \in$ AdjNodes and $\rho_j \in$ AuxTrees do
    set $p_{NL}(\eta_i)$, $p_{NR}(\eta_i)$, $p_L(\eta_i, \rho_j)$ and $p_R(\eta_i, \rho_j)$ to some initial values
end
repeat
    /* Expectation step */
    reset all counts $C_x(\eta_i, \rho_j)$ to 0
    for $\mathbf{w} \in$ TrainingSet do
        parse $\mathbf{w}$ as detailed in the Parsing section of this paper,
        keeping counts of adjunctions at each node
        /* note:  the calculation of the expression $P_{\theta_0}(\mathbf{v}, \mathbf{w})c_{ij}(\mathbf{v})$
        is described in the probability section */
        for $\eta_i \in$ LAdjNodes do
            for $\rho_j \in$ LAuxTrees do
                increment $C_{Ladj}(\eta_i, \rho_j)$ by $\sum_{\mathbf{v} \in \mathcal{V}} P_{\theta_0}(\mathbf{v}, \mathbf{w})c_{ij}(\mathbf{v})$
            end
            increment $C_{NL}(\eta_i)$ by $\sum_{\mathbf{v} \in \mathcal{V}} P_{\theta_0}(\mathbf{v}, \mathbf{w})c_{i,NL}(\mathbf{v})$
        end
        for $\eta_i \in$ RAdjNodes do
            for $\rho_j \in$ RAuxTrees do
                increment $C_{Radj}(\eta_i, \rho_j)$ by $\sum_{\mathbf{v} \in \mathcal{V}} P_{\theta_0}(\mathbf{v}, \mathbf{w})c_{ij}(\mathbf{v})$
            end
            increment $C_{NR}(\eta_i)$ by $\sum_{\mathbf{v} \in \mathcal{V}} P_{\theta_0}(\mathbf{v}, \mathbf{w})c_{i,NR}(\mathbf{v})$
        end
    end
    /* Maximization step */
    for $\eta_i \in$ LAdjNodes do
        for $\rho_j \in$ LAuxTrees do
            $\hat{p}_L(\eta_i, \rho_j) \leftarrow \frac{C_{Ladj}(\eta_i, \rho_j)}{\left(\sum_j C_{Ladj}(\eta_i, \rho_p) + C_{NL}(\eta_i)\right)}$
        end
        $\hat{p}_{NL}(\eta_i) \leftarrow 1 - \left(\sum_{\rho_j \in \text{LAuxTrees}} p_{Ladj}(\eta_i, \rho_j)\right)$
    end
    for $\eta_i \in$ RAdjNodes do
        for $\rho_j \in$ RAuxTrees do
            $\hat{p}_R(\eta_i, \rho_j) \leftarrow \frac{C_{Radj}(\eta_i, \rho_j)}{\left(\sum_j C_{Radj}(\eta_i, \rho_p) + C_{NR}(\eta_i)\right)}$
        end
        $\hat{p}_{NR}(\eta_i) \leftarrow 1 - \left(\sum_{\rho_j \in \text{RAuxTrees}} p_{Radj}(\eta_i, \rho_j)\right)$
    end
until Convergence
```

Algorithm 1: EM for PLTIG Induction

the j^{th} auxiliary tree into the i^{th} adjunction site – is the action taken to form the constituent covering $w_{s+1}, ..., w_t$ in the parse tree. This brings us to the actual model parameters θ. θ represents the set of probabilities p_{ij}, where p_{ij} is equal to the probability of α_{ij} occurring, along with $p_{i,NL}$ and $p_{i,NR}$, the probabilities that no adjunction will take place at the i^{th} adjunction site for left adjunctions and right adjunctions, respectively. $P_{\theta_0}(\mathbf{w})$ means 'probability of $\mathbf{w}$ as determined by the current parameters'. While recalculating θ, the new values of the parameters will be denoted $\hat{p}_x$. We also need to keep track of the counts $c_{ij}(\mathbf{v})$, $c_{i,NL}(\mathbf{v})$ and $c_{i,NR}(\mathbf{v})$ of the actual number of times the adjunction action α_{ij}, or no left or right adjunction, is used in derivation $\mathbf{v}$ while parsing the observed data.

Since EM is a hill-climbing algorithm, convergence is only guaranteed to occur at some *local* maximum. It is very possible for the global maximum (or maxima) to be missed. Where the algorithm converges depends greatly on the initial values of θ, and in most implementations, techniques such as random restarts are used to increase the probability that the global maximum is found. In this implementation, however, we select the initial parameter values based on a number of smoothing constants, which is described later in this paper.

4 CKY-Style Parsing

The parsing algorithm that we used for PLTIGs is a more general form of the bottom-up CKY algorithm that is commonly applied to CFGs. It is in the parsing stage, that we see the main advantage of using TIGs as our formalism as opposed to TAGs. The TIG form of this algorithm runs in time $O(n^3)$ on the length of the sentence, while for TAGs the worst case time is $O(n^6)$. Also, while both TIGs and CFGs describe the same set of context-free languages, CFGs can be seen as a special case of TIGs and thus this algorithm subsumes the CFG version (Schabes and Waters, 1995).

$$\text{(Axiom)}\quad \frac{A \rightarrow w_{i+1}}{[A, i, i+1]} \qquad \text{(Goal)}\quad \frac{[S, 0, n]}{\texttt{Complete}}$$

$$\text{(Prod)}\quad \frac{A \rightarrow BC \qquad [B, i, j] \qquad [C, j, k]}{[A, i, k]}$$

Figure 5: Deductive Rules for CFG CKY Parsing

4.1 CKY Parsing for CFGs

CKY is a bottom-up, dynamic-programming, chart-parsing algorithm; it is most commonly used to parse CFGs in Chomsky Normal Form. The algorithm works by the following inductive process. First, we create a base case to initialize the chart. We do this by adding in all the nonterminals that cover a single terminal item of the sentence. Since the grammar is in Chomsky Normal Form, we should always be able to cover every word in the sentence with this initialization. In the induction step, we try to cover each larger section of the sentence by finding a nonterminal that produces two children whose intersection is that section. For instance, if we have the rule $A \rightarrow BC$ and from the chart we know that B covers $[i, j]$ and C covers $[j, k]$, then we know A covers $[i, k]$. The complete set of rules is given in Figure (5).

4.2 CKY Parsing for TIGs

The generalized TIG CKY algorithm includes a few major changes from the CFG version. Clearly the major new case is the need to process adjunctions as well as the CFG rules. In fact, CFG rules are encompassed by TIG substitution rules and vice-versa, so adjunction is the only new case(Joshi, 1975). Another issue is the need for a TIG normal form. It is necessary to start with trees that, like Chomsky Normal Form rules, can produce any context-free language, but also are lexicalized. Finally, general TIG trees can have any finite depth, so we must extend the algorithm to process the implicit rules within each

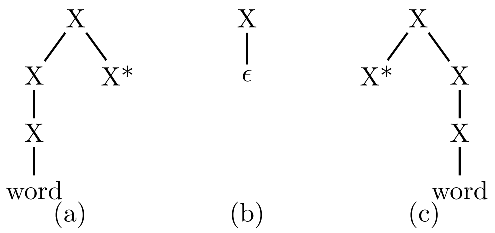


Figure 6: Normal form trees: (a) Left Auxiliary Tree; (b) Initial Tree; (c) Right Auxiliary Tree.

TIG tree.

TIG Normal Form In order to simplify the parsing process and to ensure $O(n^3)$ time, we must start with a TIG normal form. We used the style described by Hwa (2001). In her normal form, she creates two symmetrical lexical trees, one left and one right, for each entry in the vocabulary. On their non-lexical side, the trees simply end with a foot node. On their lexical side, the trees have a series of adjunction nodes terminating in the lexical item. She also creates a single initial tree, to which we can adjoin to finish the parse (see Figure (6)).

This normal form has several desirable properties. First, its trees are all binary branching. If the trees had a greater branching factor, then the algorithm would need to check more possibilities in the induction step. This problem is similar to the problem of three items on the right side of CFG rules and would cause the complexity to increase. Second, with these trees, we can parse any possible sentence. This can be shown by the simple case of repeated adjunction of right trees. If each tree contains the next word in the sentence, we can make a flat parse of any sentene. Third, the trees are both lexicalized, which takes advantage of the major feature of TIGs. Finally, neither of the trees have any substitution nodes, so we can ignore the process of looking for possible substitutions.

The CKY algorithm The data structures of the TIG chart parsing algorithm are similar to those in the CFG algorithm. In the CFG algorithm, each cell in the chart stores the nonterminal symbol that can cover the corresponding interval of the sentence. In the TIG parser, we instead store the node that covers this interval. Similarly, in place of the CFG production rules, we use both the tree structures of the normal form trees and the implicit ability to adjoin trees to adjunction nodes. In a general TIG parser we would also need to consider the ability to perform substitution, but since the normal form that we are using does not include any substitution nodes, this is not a concern.

The TIG CKY algorithm proceeds by the following process. First, we create a base case to initialize the chart. Then we go through a series of logical inferences to induce larger coverage of the sentence. Unlike with CFGs, we do not have full knowledge about a node simply because it is in the chart. In CFGs, all nonterminals in the chart can be considered the same. For TIGs, we must also keep track of whether the node is a root node, a lexical node, an adjunction node, a non-adjunction node, or a foot node. If it is an adjunction node, we need to know whether in the interval stored with the node, right or left adjunction has occurred at this node. Also we need to keep track of whether a node has been completed. If a node is completed then no more left or right adjunction will occur. In practice, for PLTIGs this information can be inferred from corresponding inside probabilities (see section 5).

To create a base case for parsing, we first add all the leaf nodes to the chart. We add each lexical leaf to the cell corresponding to its place in the sentence. This initializing step is equivalent to the base case for CFGs. We also add the foot node of each relevant tree to all the empty intervals to the right or left of its lexical node. For instance, if we add the word "hello" to the tree at the interval from i to $i+1$ then we also add the right tree foot node to the interval j to j for j less than i and the left foot tree to all the empty intervals greater than i. Even though these foot nodes do not cover anything, they must be added to the chart so that later the root node of the tree can be added to the chart. This step includes the rules LEX, LFOOT, and RFOOT Figure (7).

Once we have initialized the chart, the inheritance rules must fire before parsing can proceed. The inheritance rules are similar to the reduction rules in shift-reduce parsers for CFGs. A reduce rule infers

(LEX)

$$\frac{\texttt{lex}(\eta_A, w_{i+1})}{[\eta_A, i, i+1]}$$

(LFOOT)

$$\frac{lfoot(\eta_A) \qquad lex(\eta_B, w_j) \qquad same\_tree(\eta_A, \eta_B) \qquad k > j}{[\eta_A, k, k]}$$

(RFOOT)

$$\frac{rfoot(\eta_A) \qquad lex(\eta_B, w_j) \qquad same\_tree(\eta_A, \eta_B) \qquad i <= j}{[\eta_A, i, i]}$$

(GOAL)

$$\frac{iroot(\eta_S) \qquad [\eta_S, 0, n]}{\texttt{Finished}}$$

(COMPLETION)

$$\frac{[\eta_A, i, k, x]}{[\eta_A, i, k]}$$

(ONLY CHILD INHERITANCE)

$$\frac{[\eta_A, i, j] \qquad \eta_B \rightarrow \eta_A}{[\eta_B, i, j, \emptyset]}$$

(SIBLING INHERITANCE)

$$\frac{[\eta_A, i, j] \qquad [\eta_B, j, k] \qquad \eta_C \rightarrow \eta_A \eta_B}{[\eta_C, i, k, \emptyset]}$$

(LEFT ADJUNCTION)

$$\frac{lroot(\eta_A) \qquad adj(\eta_B) \qquad [\eta_A, i, j] \qquad [\eta_B, j, k, \emptyset]}{[\eta_B, i, k, L]}$$

(RIGHT-LEFT ADJUNCTION)

$$\frac{lroot(\eta_A) \qquad adj(\eta_B) \qquad [\eta_A, i, j] \qquad [\eta_B, j, k, R]}{[\eta_B, i, k, RL]}$$

(RIGHT ADJUNCTION)

$$\frac{rroot(\eta_A) \qquad adj(\eta_B) \qquad [\eta_A, j, k] \qquad [\eta_B, i, j, \emptyset]}{[\eta_B, i, k, R]}$$

(LEFT-RIGHT ADJUNCTION)

$$\frac{rroot(\eta_A) \qquad adj(\eta_B) \qquad [\eta_A, j, k] \qquad [\eta_B, i, j, L]}{[\eta_B, i, k, LR]}$$

Figure 7: Deductive parse rules for TIG CKY. $\emptyset$ means no adjunction has taken place, R means right adjunction, L means left adjunction, and no symbol means the node has been completed. The relation same_tree(A,B) is true if the nodes A and B are in the same tree. Lex(A, B) is true if node A is a lexical node and its lexical item is B. lfoot(A) and rfoot(A) are true if A is a left and right foot respectively, the same is true for lroot(A) and rroot(A). iroot(A) is true if A is the root of an initial tree and adj(A) is true if it is an adjunction node. The notation $A \rightarrow BC$ is used here to mean node A has children B and C.

```
Initialization step
*Sets up the base case of the chart (LEX, LFOOT, and RFOOT)*
for w ∈ Sentence do
    forall left and right lex trees with lex node w do
        Put lex node whose value in chart
        Put corresponding foot node in chart
        Call Inheritance on all nodes
    end
end
Induction step
The $O(n^3)$ algorithm (All other parse rules)
for $span = 2$ to $SentLen$ do
    for $i \leftarrow 0$ to $SentLen - span$ do
        $k \leftarrow i + span$
        for $j \leftarrow i$ to $k$ do
            forall $\eta$ that cover i-j and $\mu$ that cover j-k do
                if $\eta$ is sibling of $\mu$ then
                    chart(i,k) $\leftarrow$ parent of $\eta$
                end
                if $\eta$ is LeftRoot and $\mu$ is AdjNode then
                    if $\mu$ has no adjunctions then
                        chart(i,k) $\leftarrow$ $\mu$ with Left flag
                    end
                    if $\mu$ has a right adjunction then
                        chart(i,k) $\leftarrow$ $\mu$ with a Right-Left flag
                    end
                end
                if $\eta$ is AdjNode and $\mu$ is RightRoot then
                    if $\eta$ has no adjunctions then
                        chart(i,k) $\leftarrow$ $\eta$ with a Right flag
                    end
                    if $\eta$ has a left adjunction then
                        chart(i,k) $\leftarrow$ $\eta$ with a Left-Right flag
                    end
                end
            end
        end
        for $\eta \in$ chart(i,k) do
            Complete $\eta$
            Inherit $\eta$
        end
    end
end
```

Algorithm 2: TIG CKY parsing algorithm

from $[B, i, j]$ and $A \rightarrow B$ that $[A, i, j]$. This rule adds the adjunction nodes above each of the lexical nodes into the chart. Since CFGs have a max depth of one, the reduce rules are simple. The TIG inheritance rules must also deal with foot nodes that cover no area of the sentence and can be ignored to allow reduction. This rule adds the root node from the top adjunction node and foot node. Together the inheritance rules crawl up the trees to insure that higher nodes get in the chart. The inheritance step includes rules ONLY CHILD INHERITANCE and special cases of SIBLING INHERITANCE.

Now the main section of the algorithm begins. Starting with all the intervals of size two, the algorithm tries to parse larger intervals until it finds a parse for the whole sentence. At each interval, a node is added to the chart by any of three rules. The first rule is identical to the CFG case. This rule adds a node to the chart at $[i, k]$ if one of its children covers $[i, j]$ and the other covers $[j, k]$. The next two rules are for left and right adjunction. For right adjunction, we check to see if there is an adjunction node that covers $[i, j]$ and a right root that covers $[j, k]$. If that right root is completed, we can adjoin so that the adjunction node covers $[i, k]$. The case of left adjunction is reversed, with the left root covering $[i, j]$ and the adjunction node covering $[j, k]$. For adjunction nodes, we keep flags for which adjunctions have occured, so that we do not attempt to do more than 2 adjunctions at any node. After each round of the algorithm, we also complete each node that spans $[i, k]$. This allows the inheritance rules to fire, so that the new roots are added to the chart. We continue to apply these rules until we find a parse for $[0, n]$. At this point we know that we can parse the entire sentence. The rules in this step include the four ADJUNCTION rules, the general SIBLING INHERITANCE rule, and the COMPLETION rule.

5 Probabilities

For the E phase of the EM algorithm, we need to compute the likelihood that the parameters for the current grammar will produce the given training sentences.

We could compute the probability of every possible parse individually, but since each training sentence may have exponentially many parses, this would take exponential time. A better solution is to use dynamic programming and store reusable computations using the inside-outside algorithm, which can be run in $O(n^3)$ timeLari and Young (1990).

Applied to our domain of PLTIGs, the inside-outside algorithm defines two complementary probabilities at each pairing of a non-terminal node and a substring: an inside probability and an outside probability. An inside probability $e(\eta, i, k)$ is the probability that the node η derives the substring $w_{i+1} \dots w_k$. An outside probability $f(\eta, i, k)$ is the probability that everything outside that substring $w_{i+1} \dots w_k$ has already been derived, and only the node η is left incomplete. We store these probabilities in two charts, so that they can be computed inductively.

5.1 Inside Probabilities

For the inside probabilities, we start with a base case of substrings of length zero or one. From the base case, we can then inductively calculate the probabilities of progressively larger substrings by adjoining or combining together smaller substrings, whose probabilities have already been computed. We basically parse through the sentence from the bottom up.

This should sound familiar, as it is the CKY algorithm discussed in the previous section. The process of acquiring the inside probabilities simply entails calculating the probability for each chart entry as we step through the parsing algorithm. This section describes the calculation of the inside probability $e(\eta, i, k)$ for each chart entry $[\eta, i, k]$. The rules for calculating each probability entry mirror the parsing rules from the previous section.

We calculate $e(\eta, i, k)$ according to five cases:

- $e(\eta, i, k, \emptyset)$: η derives $w_{i+1} \dots w_k$ without any adjunctions
- $e(\eta, i, k, L)$: η derives $w_{i+1} \dots w_k$ after a left adjunctions
- $e(\eta, i, k, R)$: η derives $w_{i+1} \dots w_k$ after a right adjunctions

- $e(\eta, i, k, LR)$: η derives $w_{i+1} \ldots w_k$ after a simultaneous adjunction, and the left adjunction occurs first
- $e(\eta, i, k, RL)$: η derives $w_{i+1} \ldots w_k$ after a simultaneous adjunction, and the right adjunction occurs first

In the first case, no adjunction takes place, so there are three possible configurations:

- η is a substitution node, where ρ is the tree being substituted

$$e(\eta, i, k, \emptyset) = e(Root(\rho), i, k)$$

- η has 1 child (ONLY CHILD INHERITANCE rule)

$$e(\eta, i, k, \emptyset) = e(\eta_A, i, k)$$

- η has 2 children (SIBLING INHERITANCE rule)

$$e(\eta, i, k, \emptyset) = \sum e(\eta_A, i, j) e(\eta_B, j, k)$$

In the second case, we need to find all possible left adjunctions where the auxiliary tree, ρ_l, being adjoined can derive the left portion of the substring, $w_{i+1} \ldots w_j$, and the node η derives the rest without any further adjunctions. We must find the probabilities for all possible left adjunctions and all possible values of j. Let $\eta_A = \texttt{Root}(\rho_l)$, then we get

$$e(\eta, i, k, L) = \sum_{\rho_l} p_l(\eta, \rho_l) \sum_{j=i+1}^{k} e(\eta_A, i, j)\ e(\eta, j, k, \emptyset)$$

This reflects the LEFT ADJUNCTION rule. The first factor is the probability of adjunction for the auxiliary tree, and may be obtained from the current grammar parameters. The second and third terms are inside probabilities of smaller substrings which have already been computed.

The third case reflects the RIGHT ADJUNCTION rule is similar to the second case, with $\eta_A = \texttt{Root}(\rho_r)$

$$e(\eta, i, k, R) = \sum_{\rho_r} p_r(\eta, \rho_r) \sum_{j=i}^{k-1} e(\eta_A, j, k)\ e(\eta, i, j, \emptyset)$$

The fourth case uses the LEFT-RIGHT ADJUNCTION rule. We can think of this as a right adjunction with the constraint that a left adjunction has already taken place. This constraint has already been computed as $e(\eta, a, b, L)$ for $b - a < t - s$, so we can write the probability as

$$\begin{aligned} & e(\eta, i, k, LR) = \\ & \quad p_{lr}(\eta) \quad \textstyle\sum_{\rho_r} p_r(\eta, \rho_r) \\ & \qquad\quad \times \quad \textstyle\sum_{j=i+1}^{k-1} e(\eta_A, j, k)\ e(\eta, i, j, L) \end{aligned}$$

With the same reasoning, for the last case, the RIGHT-LEFT ADJUNCTION rule yields

$$\begin{aligned} & e(\eta, s, t, RL) = \\ & \quad p_{rl}(\eta) \quad \textstyle\sum_{\rho_l} p_l(\eta, \rho_l) \\ & \qquad\quad \times \quad \textstyle\sum_{j=i+1}^{k-1} e(\eta_A, i, j) e(\eta, j, k, R) \end{aligned}$$

To get the total inside probability, $e(\eta, i, k)$, we must normalize each of the five cases by multiplying them with the appropriate parameters.

$$\begin{aligned} e(\eta, i, k) = \quad & p_{nl}(\eta)\ p_{nr}(\eta)\ e(\eta, i, k, \emptyset) \\ + \quad & p_{nr}(\eta)\ e(\eta, i, k, L) \\ + \quad & p_{nl}(\eta)\ e(\eta, i, k, R) \\ + \quad & e(\eta, i, k, LR) \\ + \quad & e(\eta, i, k, RL) \end{aligned}$$

5.2 Outside Probabilities

The outside probabilities are calculated very similarly to the inside probabilities, but instead of starting from the bottom up, we calculate from the top down. The base case here is to start with an underived substring of length $|\mathbf{w}|$, where $\mathbf{w}$ is the training sentence. Only the empty string satisfies the outside probability where the entire sentence has not been generated. We then begin to generate the outside portions of the tree by trying the possible adjunctions or substitutions. The new outside probability is calculated using both inside and outside probabilities, which have already been computed.

Like the inside probability, outside may also be split into five cases:

1. $f(\eta, i, k, \emptyset)$: everything outside of $w_{i+1} \ldots w_k$ has been derived, and the only incomplete node η has not been adjoined.

2. $f(\eta, i, k, L)$: everything outside of $w_{i+1} \dots w_k$ has been derived, after a left adjunction on η.

3. $f(\eta, i, k, R)$: everything outside of $w_{i+1} \dots w_k$ has been derived, after a right adjunction on η.

4. $f(\eta, i, k, LR)$: everything outside of $w_{i+1} \dots w_k$ has been derived, after a simultaneous adjunction on η, with the left adjunction occurring first.

5. $f(\eta, i, k, RL)$: everything outside of $w_{i+1} \dots w_k$ has been derived, after a simultaneous adjunction on η, with the right adjunction occurring first.

For the first case, there are six possible configurations:

- η is substituted into η_p

$$f(\eta, i, k, \emptyset) = p_s(\eta_p, Tree(\eta))\ f(\eta_p, i, k)$$

- η is an only child, with parent η_p

$$f(\eta, i, k, \emptyset) = f(\eta_p, i, k)$$

- η is the left child, parent η_p and sibling η_c

$$f(\eta, i, k, \emptyset) = \sum_{j=k}^{|\mathbf{w}|} f(\eta_p, i, j)\ e(\eta_c, k, j)$$

- η is the right child, parent η_p and sibling η_c

$$f(\eta, i, k, \emptyset) = \sum_{j=0}^{i} f(\eta_p, j, k)\ e(\eta_c, j, i)$$

- η is the root of a left auxiliary tree, adjoined to η_p

$$\begin{aligned} f(\eta, i, k, \emptyset) = & \\ & \sum_{\eta_p} p_l(\eta_p, \rho_l) \\ & \textstyle\sum_{j=k}^{|\mathbf{w}|} \quad f(\eta, i, j, \emptyset)\ [\quad e(\eta_p, k, j, \emptyset)\ P_{nr}(\eta_p) + \\ & \qquad\qquad\qquad\qquad e(\eta_p, k, j, R)\ P_{rl}(\eta_p)\ ] \end{aligned}$$

- η is the root of a right auxiliary tree, adjoined to η_p

$$\begin{aligned} f(\eta, i, k, \emptyset) = & \\ & \sum_{\eta_p} p_r(\eta_p, \rho_r) \\ & \textstyle\sum_{j=0}^{i} \quad f(\eta_p, j, k, \emptyset)\ [\quad e(\eta_p, j, i, \emptyset) P_{nl}(\eta_p) + \\ & \qquad\qquad\qquad\qquad e(\eta_p, j, i, L) P_{lr}(\eta_p)\ ] \end{aligned}$$

In the left adjunction case, we must consider all possible left adjunctions and all breakpoints j, between 0 and i, such that the left auxiliary tree, ρ_l we adjoin will cover substring $w_j \dots w_i$.

$$\begin{aligned} f(\eta, i, k, L) = & \textstyle\sum_{\rho_l} p_l(\eta, \rho_l) \\ \times & \textstyle\sum_{j=0}^{i} e(\texttt{Root}(\rho_l), j, i)\ f(\eta, j, k, \emptyset) \end{aligned}$$

The right adjunction case is:

$$\begin{aligned} f(\eta, i, k, R) = & \textstyle\sum_{\rho_r} p_r(\eta, \rho_r) \\ \times & \textstyle\sum_{j=k}^{|\mathbf{w}|} e(\texttt{Root}(\rho_r), k, j)\ f(\eta, i, j, \emptyset) \end{aligned}$$

For the simultaneous adjunction cases, we again think of them as single adjunctions, with the constraint that the other adjunction has already taken place.

$$\begin{aligned} & f(\eta, i, k, LR) = \\ & \quad p_{lr}(\eta) \quad \textstyle\sum_{\rho_l} p_l(\eta, \rho_l) \\ & \qquad \times \quad \textstyle\sum_{j=0}^{i+1} e(\texttt{Root}(\rho_l), j, i)\ f(\eta, j, k, R) \end{aligned}$$

$$\begin{aligned} & f(\eta, i, k, RL) = \\ & \quad p_{rl}(\eta) \quad \textstyle\sum_{\rho_r} p_r(\eta, \rho_r) \\ & \qquad \times \quad \textstyle\sum_{j=k}^{|\mathbf{w}|} e(\texttt{Root}(\rho_r), k, j)\ f(\eta, i, j, L) \end{aligned}$$

For the total outside probability, we again have to normalize the 5 components:

$$\begin{aligned} f(\eta, i, k) = & \quad p_{nl}(\eta)\ p_{nr}(\eta)\ f(\eta, i, k, \emptyset) \\ + & \quad p_{nr}(\eta)\ f(\eta, i, k, L) \\ + & \quad p_{nl}(\eta)\ f(\eta, i, k, R) \\ + & \quad f(\eta, i, k, LR) \\ + & \quad f(\eta, i, k, RL) \end{aligned}$$

5.3 Calculating Expected Counts for EM

This brings us back to the E-step of the EM algorithm (Algorithm 1). With the above probabilities, we are now able to calculate the expected adjunction counts for each training sentence based on the parse of that sentence and the current model parameters θ_0.

For each training sentence $\mathbf{w}$, we have to calculate four sets of counts: $C_{Ladj}(\eta_i, \rho_j)$, $C_{NL}(\eta_i)$, $C_{Radj}(\eta_i, \rho_j)$ and $C_{NR}(\eta_i)$. These are derived as shown in Figure (8), where p_X refers to a parameter within θ_0.

6 Implementation

In this section, we describe a few details of our implementation.

6.1 Initialization

In order to initialize our learning algorithm, we must assign initial values to the parameters of the elementary trees. Recall that there will be one left auxiliary and one right auxiliary tree for each word in the training corpus, and that each of these trees will have two adjunction nodes. The most interesting parameters are the probability distributions for left and right adjunction at each of the adjunction sites. We can control the types of grammars that are generated by setting no-adjunction probabilities to 1.0 and thus preventing adjunctions at specific sites. In our experiments, we set the no-adjunction probabilities according to Table 1 where $|\mathcal{W}|$ is the number of words in the training corpus, and k is equal to

$$\frac{1-\delta}{|\mathcal{W}|+1} \quad .$$

Site	No Adjunction
Upper Left	1.0
Upper Right	k
Lower Left	k
Lower Right	k

Table 1: Initial non-adjunction probabilities for auxiliary trees.

We use the parameter δ to make the probability of adjoining any particular tree slightly higher than the no-adjunction probability. In our system, δ is set manually based on several smoothing parameters to teh value 0.0079.

6.2 Thresholding

In practical situations, probabilistic parsers tend to be hindered by a large number of possible sub-parses that have a very low probability. While this does not effect the worst-case complexity, it can cause a constant time slowdown of the algorithm. To combat this problem, we used two methods of thresholding discussed in Goodman (1997) to remove unlikely parses.

The first is a simple absolute minimum method. Before starting the parse, we select a lower bound for all probabilities and adjust this bound to the length of the sentence. If at any point, the parser tries to include a node with a probability below this minimum that probability is immediately set to zero.

The second method is known as Beam Search Thresholding. In this method, the thresholding minimum is set relative to each cell in the chart. Before starting the parse, we select the maximum ratio between the highest and lowest value probability in any cell. As we add nodes to each cell, we compute the highest probability seen so far. Then in the completion step of the algorithm, we eliminate any nodes that fall below the alloted ratio.

The problem with both these methods is that by using them we lose both the completeness property of CKY and the optimality property of the inside-outside algorithm. These methods could prune away the best parse or even all the parses. We can get back the completeness property by using an iterative search. We start by pruning a large amount of nodes, and then lessen the thresholding until we get a valid parse. Unfortunately, we can never ensure optimality using this type of thresholding.

6.3 Python

We wrote our training and parsing system in Python as opposed to the Hwa (2001) system, that was constructed in C. Python is an interpreted, object-oriented programming language freely available at

$$\begin{aligned}
C_{Ladj}(\eta_i,\rho_j) = & \sum_{\mathbf{v}\in\mathcal{V}} P_{\theta_0}(\mathbf{v},\mathbf{w})c_{ij}(\mathbf{v}) \\
& = p_L(\eta_i,\rho_j) \sum_{s=0}^{|\mathbf{w}|-1} \sum_{t=s+1}^{|\mathbf{w}|} f(\eta_i,s,t,\emptyset) \sum_{r=s+1}^{t} e(Root(\rho_j),s,r) \left(\begin{smallmatrix} p_{NR}(\eta_i)e(\eta_i,r,t,\emptyset) \\ +p_{RL}(\eta_i)e(\eta_i,r,t,R) \end{smallmatrix} \right) \\
C_{NL}(\eta_i) = & \sum_{\mathbf{v}\in\mathcal{V}} P_{\theta_0}(\mathbf{v},\mathbf{w})c_{i,NL}(\mathbf{v}) \\
& = p_{NL}(\eta_i,\rho_j) \sum_{s=0}^{|\mathbf{w}|-1} \sum_{t=s+1}^{|\mathbf{w}|} f(\eta_i,s,t,\emptyset) \left(\begin{smallmatrix} p_{NR}(\eta_i)e(\eta_i,r,t,\emptyset) \\ +p_{RL}(\eta_i)e(\eta_i,r,t,R) \end{smallmatrix} \right) \\
C_{Radj}(\eta_i,\rho_j) = & \sum_{\mathbf{v}\in\mathcal{V}} P_{\theta_0}(\mathbf{v},\mathbf{w})c_{ij}(\mathbf{v}) \\
& = p_R(\eta_i,\rho_j) \sum_{s=0}^{|\mathbf{w}|-1} \sum_{t=s+1}^{|\mathbf{w}|} f(\eta_i,s,t,\emptyset) \sum_{r=s+1}^{t} e(Root(\rho_j),s,r) \left(\begin{smallmatrix} p_{NL}(\eta_i)e(\eta_i,r,t,\emptyset) \\ +p_{LR}(\eta_i)e(\eta_i,r,t,L) \end{smallmatrix} \right) \\
C_{NR}(\eta_i) = & \sum_{\mathbf{v}\in\mathcal{V}} P_{\theta_0}(\mathbf{v},\mathbf{w})c_{i,NR}(\mathbf{v}) \\
& = p_{NR}(\eta_i,\rho_j) \sum_{s=0}^{|\mathbf{w}|-1} \sum_{t=s+1}^{|\mathbf{w}|} f(\eta_i,s,t,\emptyset) \left(\begin{smallmatrix} p_{NL}(\eta_i)e(\eta_i,r,t,\emptyset) \\ +p_{LR}(\eta_i)e(\eta_i,r,t,L) \end{smallmatrix} \right)
\end{aligned}$$

Figure 8: Expected Adjunction Count Calculations

http://www.python.org/. We decided to use Python for several reasons:

- Python's object-oriented structure will allow us to preserve data structures when attempting our stated goal of extending this project into a synchronous parser.
- Using Python allowed us to leverage the structures written as part of the Natural Language Toolkit (NLTK). We used NLTK classes to interact with the Penn Tree Bank, as base classes for our TIG trees, and to handle some of the frequency distributions.
- One of our implementation goals was to structure the program so that the parse rules would be apparent to anyone reading the code. We feel that the straightforward syntax of Python makes the implementation more readable and modular.

The major downside of using an interpreted language like Python over C is the speed difference. Even with extra optimizations, the Python code runs significantly slower than the equivalent C code.

6.4 Penn Tree Bank

In our implementation, we used the Penn TreeBank Wall Street Journal corpus for training information. The WSJ corpus includes sentences from the Wall Street Journal that are skeletally parsed for syntactic structure. We used these sentences as the training sentences for the parser. Since the sentences include parse information, we had our parser ignore any constituents that were not marked in the hand-parsed data. This step acts as the supervised learning section of the algorithm.

7 Conclusion

We started this project with the goal of writing a synchronous TAG parser with trees from a bilingual dictionary. Our aim was to use this synchronous parser for statistical machine translation(MT), in the method described in Melamed (2003). We hypothesized that by starting with synchronous tree rules that were prealigned and then learning adjunction probabilities from a bilingual corpus, we could improve upon existing statistical MT methods.

Creating a synchronous TAG parser is still our final goal, but there are some major steps. We now have a monolingual TIG parser based on a set of simple normal form rules. For our next step, we would like to generalize this process to synchronous TIGs in a similar normal form. We predict that this will involve generalizing the CKY algorithm to perform adjunctions simultaneously on linked nodes. This concept is being explored concurrently by Nesson and Ganatra (2005).

Assuming that we can extend to a synchronous normal form, there are still some major questions outstanding about future directions. One linguistic question is whether the extra expressiveness of TAG can be utilized in spite of its greater complexity. Additionally, it is not clear if the extra expressiveness is worth the additional cost. Another question is how we can extend the parsing algorithm described in this paper to parse more complicated elementary trees while retaining its computational tractability.

References

NM Laird AP Dempster and DB Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society B*, 39(1):1–38, 1977.

Peter F. Brown, John Cocke, Stephen Della Pietra, Vincent J. Della Pietra, Frederick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. A statistical approach to machine translation. *Computational Linguistics*, 16(2):79–85, 1990.

Joshua Goodman. Global thresholding and multiple-pass parsing. In Claire Cardie and Ralph Weischedel, editors, *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 11–25. Association for Computational Linguistics, Somerset, New Jersey, 1997.

Rebecca Hwa. *Learning Probabilistic Lexicalized*

Grammars for Natural Language Processing. PhD thesis, Harvard University, 2001.

Aravind Joshi and Yves Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69–124. Springer, New York, NY, 1997.

Aravind K. Joshi. Tree adjunction grammars. *Journal of Computer and System Sciences*, 10(1), 1975.

K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

I. Dan Melamed. Multitext grammars and synchronous parsers. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 79–86, Edmonton, Alberta, Canada, May-June 2003.

Rebecca Nesson and Sheel Ganatra. Something about synchronous tig parsing. note, 2005.

Yves Schabes and Richard C. Waters. Tree insertion grammar: a cubic-time, parsable formalism that lexicalizes context-free grammar without changing the trees produced. *Fuzzy Sets Syst.*, 76(3):309–317, 1995. ISSN 0165-0114.

Stuart Shieber and Yves Schabes. Synchronous tree adjoining grammars. In *13th International Conference on Computational Linguistics*, volume 3, pages 1–6, 1990.

Warren Weaver. Translation. In William N. Locke and A. Donald Booth, editors, *Machine Translation Of Languages: Fourteen Essays*, chapter 1, pages 15–23. The Technology Press of The Massachusetts Institute of Technology and John Wiley & Sons, Inc., New York, NY, 1955.