Bidirectional LTAG Dependency Parsing

Libin Shen
BBN Technologies

Aravind K. Joshi
University of Pennsylvania

We propose a novel algorithm for bi-directional parsing with linear computational complexity, and apply this algorithm to LTAG dependency parsing, revealing deep relations which are unavailable in other approaches and difficult to learn. We have evaluated the parser on the LTAG-spinal Treebank. Experimental results show a significant improvement over the incremental parser described in (Shen and Joshi 2005b). This learning algorithm can be generalized as graph-based incremental construction for other structure prediction problems in NLP.

Lexicalized Tree Adjoining Grammar (LTAG) is a grammar which has attractive properties both from the points of view of linguistic description and Natural Language Processing (NLP). A recent review of TAG is given in (Joshi and Schabes 1997), which provides a detailed description of TAG with respect to linguistic, formal, and computational properties.

LTAG has appropriate generative capacities both weak and strong, the latter being more important linguistically. Processing over deep structures in the LTAG representation correlates well with natural language studies.

In LTAG, each word is associated with a set of *elementary trees*. Each elementary tree represents a possible tree structure for the word. There are two kinds of elementary trees, *initial trees* and *auxiliary trees*. Elementary trees can be combined through two operations, *substitution* and *adjunction*. Substitution is used to attach an initial tree, and adjunction is used to attach an auxiliary tree.

In this article, we are interested in the so called LTAG dependency parsing. By LTAG dependency, we mean the dependency relation of the words encoded in an LTAG derivation tree. Compared to the dependency relations defined over Context-Free Grammars (Magerman 1995), LTAG dependency reveals deeper relations which are more useful for NLP applications, thanks to the extended domain of locality in LTAG. It should be noted that LTAG dependencies could be non-projective dependencies due to *wrapping adjunction*.

For the purpose of building an efficient LTAG dependency parser, we will propose a novel parsing strategy as well as a new learning algorithm which searches for the LTAG dependency tree in a bidirectional style. We will first describe the idea of our approach in Section 1. In Section 2, we will use a detailed example to illustrate our algorithm. The formal definition of the algorithms will be given given in Section 3. In Section 4, we will illustrate the details of the bidirectional LTAG dependency parsing. In Section 5, we will compare the novel algorithm to other related works, and in Section 6, we will report the experiments of dependency parsing on the LTAG-spinal Treebank.

1. Idea

1.1 Parsing Algorithms for Tree Adjoining Grammar

Vijay-Shanker and Joshi (1985) introduced the first TAG parser in a CYK-like algorithm. Because of the adjoining operation, the time complexity of LTAG parsing is as large as $O(n^6)$, compared with $O(n^3)$ of CFG parsing, where n is the length of the sentence to be parsed.

Schabes and Joshi (1988) introduced an Earley style TAG parser that utilized top-down prediction to speed up parsing. Lavelli and Satta (1991) designed a bidirectional CYK-like parser for LTAG, which used head-driven lexical information for prediction. van Noord (1994) proposed a similar bidirectional algorithm, the *head-corner* parser, which handles both substitution and adjunction.

However, the high time complexity prevents CYK-like LTAG parsing from real-time applications (Sarkar 2000).

Furthermore, the treatment of coordination structure was not incorporated in the original definition of LTAG. Sarkar and Joshi (1996) proposed a method of generating coordination structures dynamically in parsing. This further increases the complexity of LTAG parsing.

Variants of TAG have been proposed to decrease the computational complexity. A popular simplification of TAG is the Tree Insertion Grammar (TIG) (Schabes and Waters 1995; Chiang 2000). Since TIG does not allow *wrapping adjunction*, its time complexity of parsing is just $O(n^3)$. However, TIG has a much weaker generative power than TAG both in the strong and the weak senses.

In (Shen and Joshi 2005a), we worked on LTAG-spinal, an interesting subset of LTAG, which preserves almost all of the strong generative power of LTAG, and it is both weakly and strongly more powerful than CFG. The LTAG-spinal formalism also makes it easy to incorporate the coordination operation in parsing. In that paper, we presented a effective and efficient statistical incremental parsing for LTAG-spinal. Since left-to-right beam search is used, its computational complexity is proportional to the length of a sentence, or $O(n)$.

We notice that the strong ambiguity of the LTAG operations makes it hard to maintain different parses with a beam search from left to right. With a beam width of 10, we usually end up with similar parses. Therefore, it is impossible to maintain in the beam the partial structures which is undesirable with respect to a local context, but preferable according to long distance relations.

This problem can be alleviated via introducing bidirectional search strategy in linear parsing. We can effectively utilize all the strong indicator across the whole sentence to build partial structures, so as to make the use of long distance relations possible.

A similar search strategy is employed in the chunking-attachment framework (Abney 1991; Joshi and Hopely 1997) of parsing. A chunker first recognizes chunks in an input sentence, and an attacher builds a parse tree with chunks. In this way, chunks can be viewed as the segments that we are the most confident of in an input sentence. Chunks serve as the shared structure in this two-step parser.

Here, we are going to employ bidirectional search strategy in linear parsing for LTAG, in the context of statistical parsing. Furthermore, we are especially interested in dependency relation encoded in LTAG derivation trees, which reveals deeper and non-projective relations which are lack in previous works (Yamada and Matsumoto 2003; McDonald, Crammer, and Pereira 2005) on English dependency parsing on the Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1994). As the future work, we can expand an LTAG dependency tree to an LTAG derivation tree, and further incorporate semantic parsing (Zettlemoyer and Collins 2005) into this LTAG based platform.

1.2 Parsing as Search

Parsing can be modeled as a search problem. Klein and Manning (2003) proposed an A* search algorithm for PCFG parsing. The score of a partial parse is the sum of two part, *a* and *b*, where *a* is the estimated score for the *outside* parse and *b* is the actual score for the *inside* parse. The outside estimate is required to be *admissible* for A* search, which means it should be higher than the actual score. Two methods were proposed to estimate the outside score, One is based on the input information in a local context, e.g. POS tags of nearby words. The other is to utilize the score (log-likelihood) given by a simplified grammar.

In fact, incremental parsing can also be viewed as a search problem, in which path selection is to some extent fixed. The Perceptron like algorithm proposed in (Collins and Roark 2004) is a learning algorithm specially designed for the problems of this class. They also improved this learning algorithm by introducing *early update*, which makes the search in training stop when the gold standard parse is not in the beam. This algorithm has been successfully applied to incremental parsing for CFG.

Daumé III and Marcu (2005) generalized this Perceptron like learning algorithm to general search problems. In their framework, the score of a path is also composed of two parts, *g* and *h*. Here *g* is the score of the path component, computed as a linear function of features as in Perceptron learning, and *h* is the estimate of the heuristic component, which is similar to A* search [1]. What makes it different from Perceptron with *early update* is that a set of gold-standard compatible paths are introduced into the queue of candidate paths when there is no gold-standard compatible path in the queue, instead of making the search stop. This algorithm has been successfully employed in applications like chunking and POS tagging, but it has not been used in parsing yet.

Here, we are going to explore the greedy search mechanism and the Perceptron learning algorithm for the bidirectional parsing strategy in the context of LTAG parsing.

In our recent work (Shen, Satta, and Joshi 2007), we applied a bidirectional Perceptron learning algorithm to POS tagging, and achieved an accuracy of 97.33% on the standard PTB test set. The previous best result in literature on this data set is 97.24% (Toutanova et al. 2003). We used fewer features than what were used in (Toutanova et al. 2003) to achieve our result[2]. Here, we are going to extend the this bidirectional Perceptron learning algorithm from sequence labeling to LTAG dependency parsing.

1.3 Our approach

In the algorithms previously used in left-to-right search, Hypotheses are always for the prefixes of a sentences. Therefore, most of the competing hypotheses are always incompatible with each other, and there is no good way to employ shared structures over hypotheses over partial structures like in CYK parsing.

If we use bidirectional search to build hypotheses on non-overlapping segments over a sentence, we can easily employ shared structures. Intuitively, we can take advantage of nearby hypotheses for the estimation of the outside score, so that the overall score for each hypothesis will be more accurate.

1 *h* is not required to be *admissible* here.

2 In (Toutanova et al. 2003), one of the features was defi ned on the output of a simple named entity recognizer, and it helped to obtain signifi cant improvement on the accuracy. Since it is diffi cult to duplicate exactly the same feature for comparison, we did not use this feature, although it is easy to for us to use complicated features like this in our learning algorithm.

In this way, we can find the global hypothesis more efficiently. This method will be especially useful for greedy search. However, this approach results in the difficulty of maintaining a hypothesis with its context hypotheses, for which we will provide a solution later in this article (Section 3). On the other hand, we still want to make the computational complexity linear to the length of a sentence.

As far as LTAG dependency parsing is concerned, we need to design some data structures to represent the partial result of an LTAG dependency tree, so that we can build partial dependency trees step by step. As we have noted, LTAG dependency could be a non-projective relation. So we will use the mechanism of *visibility* which we will explain in detail in Section 4.

In this article, we will use LTAG-spinal, a variant of traditional LTAG proposed in (Shen and Joshi 2005a) for the purpose of statistical processing. In LTAG-spinal, substitution and non-predicate adjunction are merged as **attachment**, so as to encode the ambiguity of argument-adjunct distinction, while **adjunction** is reserved for adjunctions of raising verbs, i.e. *seem*, and passive Exceptional Case Marking (ECM) verbs, i.e. *expected*, which may result in wrapping structures. Therefore, LTAG-spinal allows non-projective dependencies to reveal deep relations, which makes it different from other simplifications of LTAG, like Tree Insertion Grammar.

We will employ the LTAG-spinal Treebank reported in (Shen and Joshi 2005a) for training and evaluation. The LTAG-spinal Treebank was extracted from the Penn Treebank reconciled with Propbank annotations (Palmer, Gildea, and Kingsbury 2005). We simply use the dependency relations defined on the LTAG-spinal derivation trees. It should be noted that, in the LTAG-spinal Treebank, predicate coordination are explicitly represented. We will introduce a special adjunction operation called **conjunction** to build predicate coordination incrementally.

For detailed description the LTAG-spinal formalism and the LTAG-spinal Treebank, see (Shen and Joshi 2005a). In order to provide a quick reference, we attach an Appendix of LTAG and LTAG-spinal to the end of the article.

2. An Example

In this section, we show the data structures and the bidirectional dependency parsing algorithm with an example, and leave the formalization to the next section.

Initialization

Suppose the input sentence is as follows.

Example 1
graduate students were not expected to take ten courses previously

Each word is associated a set of hypothesis POS tags in the input, as shown in Figure 1. For initialization, each word comprises a **fragment**, a continuous part of a sentence. A POS tag with the lexical item is called a **node** in dependency parsing. A node is a unit structure of analysis for further operations. For initialization, each node comprises a **fragment hypothesis**, which represents a possible analysis for a fragment. Due to the limitation of space, we ignore the lexical item in a node in Figure 1.

Step 1

We can combine the hypotheses for two nearby fragments with various operations like attachment and adjunction. For example, we can attach *JJ(graduate)* to *NNS(students)*, which is compatible with the gold standard, or adjoin *VB(take)* to *CD(ten)*, which is incompatible with the gold

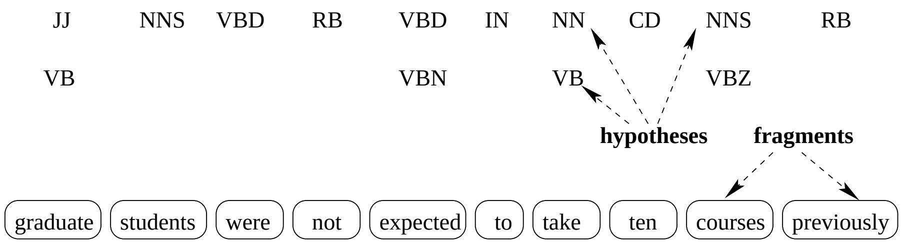

Figure 1
Initialization

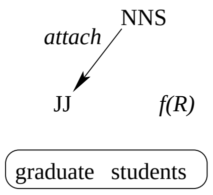

Figure 2
To attach *JJ(graduate)* to *NNS(students)*

standard. We can represent an **operation** $R_{type,main}$ with a 4-tuple

$$R_{type,main}(f_l, f_r, n_l, n_r), \tag{1}$$

where *type* $\in \{adjunction, attachment, conjunction\}$, is the type of the operation. *main* = *left* or *right*, representing whether the left or the right tree is the main tree. f_l and f_r stand for the left and right fragment hypotheses involved in the operation. n_l and n_r stand for the left and right nodes involved in the operation.

Suppose we have an operation R on fragment hypotheses $R.f_l$ and $R.f_r$[3], we generate a new hypotheses $f(R)$ for the new fragment which contains the fragments of both $R.f_l$ and $R.f_r$. For example, Figure 2 shows the result of attaching *JJ(graduate)* to *NNS(students)*. The new fragment hypothesis $f(R)$ is for the new fragment *graduate students*.

We use a priority queue Q to store all the candidate operations that could be applied to the current partial results. Operations in Q are ordered with the score of an operation, $s(R)$. We have

$$s(R) = \mathbf{w} \cdot \Phi(R) \tag{2}$$
$$score(f(R)) = s(R) + score(R.f_l) + score(R.f_r), \tag{3}$$

where $s(R)$ is the score of the operation R, which is calculated as the dot product of a weight vector $\mathbf{w}$ and $\Phi(R)$, the feature vector of R. $s(R)$ is used to order the operations in Q.

The **feature** vector $\Phi(R)$ is defined on $R.f_l$ and $R.f_r$, as well as the context hypotheses. If $\Phi(R)$ only contains information in $R.f_l$ and $R.f_r$, we call this **level-0 feature dependency** [4].

3 $R.f_l$ and $R.f_r$ represent f_l and f_r of R respectively.
4 The dependency in *feature dependency* is different from the dependency in *dependency parsing*. For example, as shown in Figure 3, the feature dependency is between the hypotheses, while the dependency of interest is between the words with POS tags

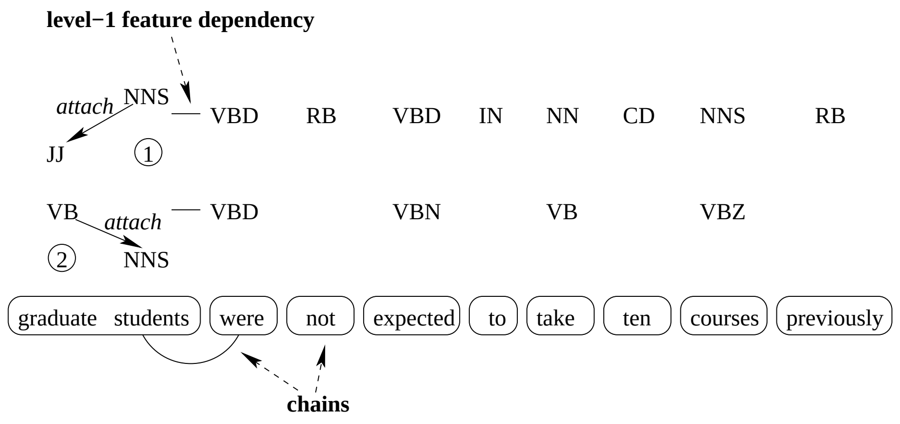

Figure 3
Step 1 : to combine *graduate* and *students*

However, we may want to use the information in nearby fragment hypotheses in some cases. For example, for the operation of attaching *JJ(graduate)* to *NN(student)*, we can check whether the root node of the hypothesis for the fragment containing *were* is a verb. We can define a feature for this, and this feature will be a strong indicator of the attachment operation. If features contain information of nearby fragment hypotheses, we call this **level-1 feature dependency**. By introducing level-1 feature dependencies, we actually calculate the score of a hypothesis by exploiting the information of outside hypotheses, as we have proposed earlier. Throughout this example, we will use level-1 feature dependency.

Suppose the operation of attaching *JJ(graduate)* to *NNS(students)* has the highest score, which is conditioned on the context that the POS tag for *were* is VBD. *were* is a nearby fragment. Therefore we need a data structure to maintain this relation. As shown in Figure 3, we introduce a **chain** which consists of two fragments, *graduate students* and *were*. In general, we use a chain to represent a set of fragments, such that hypotheses of each fragment always have feature dependency relations with some other hypotheses of some fragments within the same chain. Furthermore, each fragment can only belong to one chain. By default, each stand-alone fragment also comprises a chain, as shown in Figure 3.

Intuitively, if we select one fragment hypothesis of a segment, we must select a chain of other hypotheses by recursively using the feature dependency relation. The chain structure is used to store the fragments where the related hypotheses come from. A set of related fragment hypotheses is called a **chain hypothesis**. For a given chain, each fragment contributes a fragment to build a chain hypothesis. Here, each two fragment hypotheses of nearby fragments have feature dependency relation.

Suppose we use beam search and set beam width to two for each chain, which means that we keep the top two chain hypotheses for each chain. Figure 3 shows two chain hypotheses for the chain of *graduate students* - *were*. For the chain *graduate student* - *were*, each chain hypothesis consists of two fragment hypotheses respectively.

The score of a chain hypothesis is the sum of the scores of the fragment hypotheses in this chain hypothesis. For chain hypothesis c, we have

$$score(c) = \sum_{\text{fragment hypothesis } f \text{ of } c} score(f) \tag{4}$$

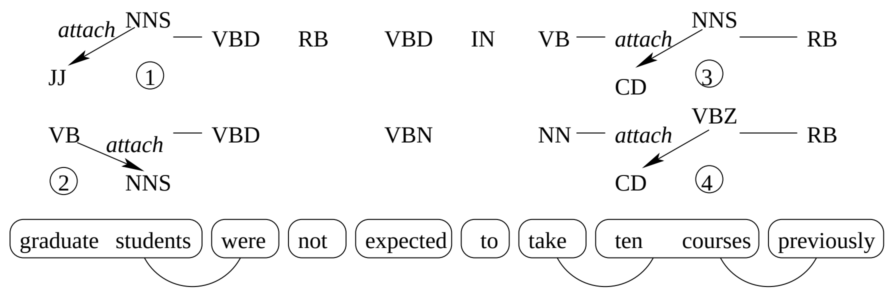

Figure 4
Step 2 : to combine *ten* and *courses*

It is easy to see that

- *hypotheses for the same chain are mutually exclusive*, and
- *hypotheses for different chains are compatible with each other*

By saying that two partial hypotheses are compatible, we mean that there exists a global hypothesis which contains these two hypotheses.

For ease of the description later in this section, we assign unique IDs, 1 and 2, to the two fragment hypotheses for *graduate students*, as shown in Figure 3.

After building the new chain and its hypotheses, we update the queue of candidate operations *Q*.

Step 2

Suppose the next operation with the highest score is to attach *CD(ten)* to *NNS(courses)* under the context of *VB(take)* and *RB(previously)*, generating hypothesis 3, as shown in Figure 4. So we generate a new fragment, *ten courses*, and build a new chain containing *take - ten courses - previously*. Now, both NP chunks have been recognized.

Step 3

Suppose the next operation with the highest score is to attach *to* to *take* under the context of *VBN(expected)* and hypothesis 3, generating hypothesis 5, as shown in Figure 5. The chain *take - ten courses - previously* grows leftwards. We still keep top two chain hypotheses. The second best operation on the two fragments is to attach *to* to *take* under the context of *VBD(expected)* and hypothesis 3, generating hypothesis 6.

Step 4

Suppose the next operation is to adjoin *VBN(expected)* to *VB(take)* in hypothesis 5, generating hypothesis 7, as shown in Figure 6.

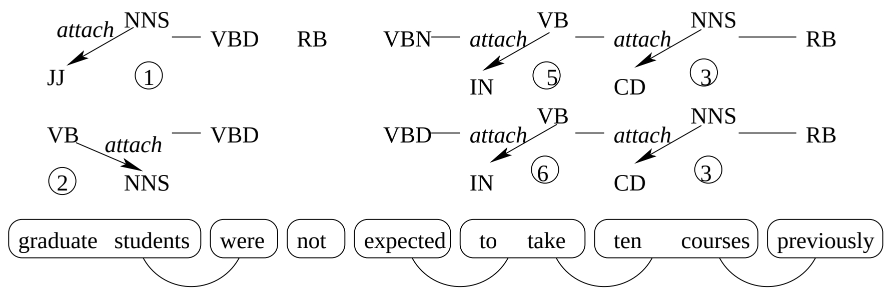

Figure 5
Step 3 : to combine *to* and *take*

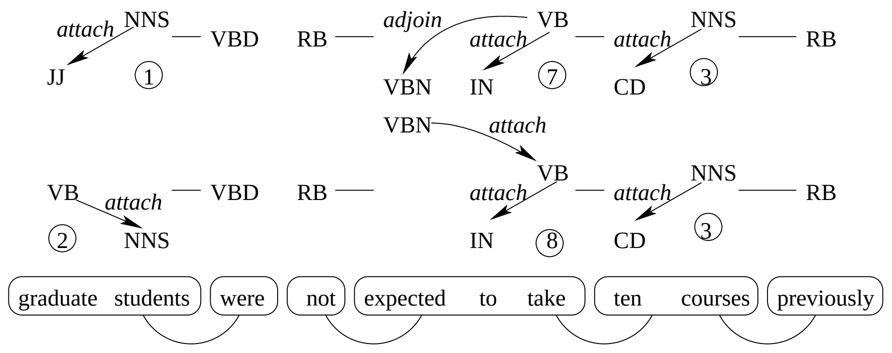

Figure 6
Step 4 : to combine *expected* and *to take*

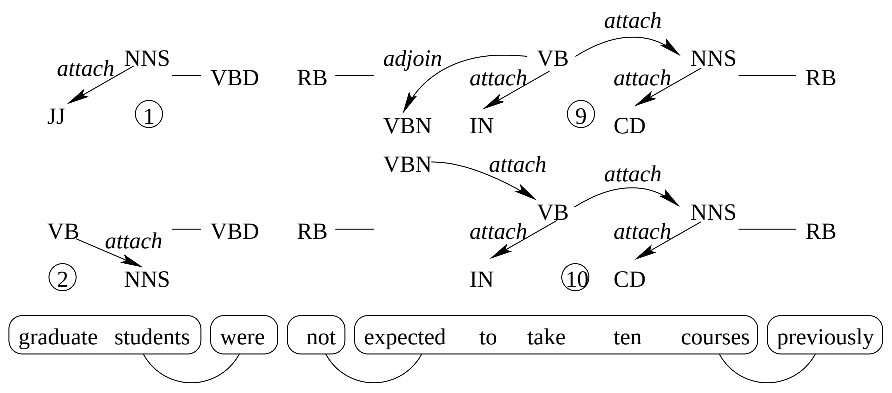

Figure 7
Step 5 : to combine *expected to take* and *ten courses*

Step 5

Suppose the next operation with the highest score is to attach *NNS(courses)* in hypothesis 3 to *VB(take)* in hypothesis 7, generating hypothesis 9, as shown in Figure 7.

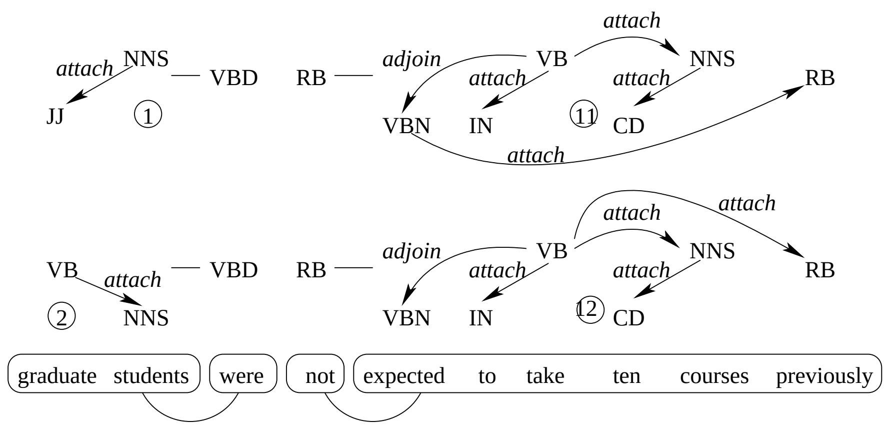

Figure 8
Step 6 : to combine *expected to take ten courses* and *previously*

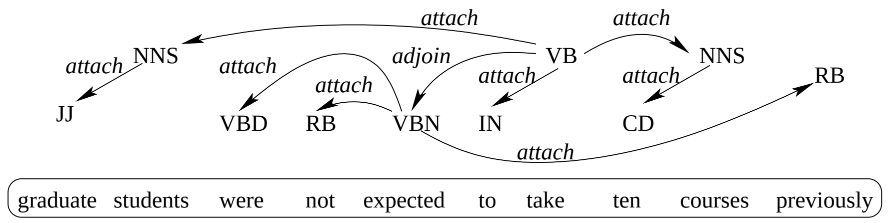

Figure 9
Final output

Step 6

Suppose the next operation with the highest score is to attach *previously(RB)* to *VBN(expected)* in hypothesis 9, generating hypothesis 11. Here, the node *VBN(expected)* is *visible* to *ADV(previously)*.

It should be noted that, this operation results in a non-projective relation, because *take* is between *expected* and *previously*. We will explain how wrapping structures like this are generated in detail in Section 4.1, in which the *spinal adjunction* property of the LTAG-spinal Treebank will be utilized.

Final Output

We repeat combining hypotheses until there is only one fragment which ranges over the whole input sentence. Then we output the parse with the highest score, as shown in Figure 9.

3. Data Structures and Algorithms

Now we define the algorithm formally. Instead of giving an algorithm specially designed for parsing, we generalize the problem for graphs. A sentence can be viewed as a linear graph composed of a sequence of vertices such that adjacent vertices in the sequence are adjacent in the graph. We define the data structures in Section 3.1. In Sections 3.2 and 3.3, we present Perceptron like search and training algorithms respectively.

3.1 Data Structures

We are given a connected graph $G(V,E)$ whose hidden structure is U, where vertices $V = \{v_i\}$, edges $E \subseteq V \times V$ is a symmetric relation, and $U = \{u_k\}$ is composed of a set of elements that vary with applications. As far as dependency parsing is concerned, the input graph is simply a linear graph, where $E(v_{i-1}, v_i)$. As to the hidden structure, $u_k = (v_{s_k}, v_{e_k}, b_k)$, where vertex v_{e_k} depends on vertex v_{s_k} with label b_k.

A graph-based incremental construction algorithm looks for the hidden structure with greedy search in a bottom-up style.

Let x_i and x_j be two sets of connected vertices in V, where $x_i \cap x_j = \emptyset$ and they are directly connected via an edge in E. Let y^{xi} be a hypothesized hidden structure of x_i, and y^{xj} a hypothesized hidden structure of x_j.

Suppose we choose to combine y^{xi} and y^{xj} with an operation R to build a hypothesized hidden structure for $x_k = x_i \cup x_j$. We say the process of construction is **incremental**[5] if the output of the operation, $y^{xk} = R(x_i, x_j, y^{xi}, y^{xj}) \supseteq y^{xi} \cup y^{xj}$ for all the possible x_i, x_j, y^{xi}, y^{xj} and operation R. As far as dependency parsing is concerned, incrementality means that we cannot remove any links coming from the substructures.

Once y^{xk} is built, we can no longer use y^{xi} or y^{xj} as a building block in the framework of greedy search. It is easy to see that left to right incremental construction is a special case of our approach. So the question is how to decide the order of construction as well as the type of operation R. For example, in the very first step of dependency parsing, we need to decide which two words are to be combined as well as the dependency label to be used.

This problem is solved statistically, based on the features defined on the substructures involved in the operation and their context. Given the weights of these features, we will show in the next section how these weights guide us to build a set of hypothesized hidden structures with beam search. In Section 3.3, we will present a Perceptron like algorithm to obtain the weights.

Now we formally define the data structure to be used in our algorithms. Most of them were previously introduced in an informal way.

A **fragment** is a *connected* sub-graph of $G(V,E)$. Each fragment x is associated with a set of hypothesized hidden structures, or **fragment hypotheses** for short: $Y^x = \{y^x_1, ..., y^x_k\}$. Each y^x is a possible fragment hypothesis of x.

It is easy to see that an operation to combine two fragments may depend on the fragment hypotheses in the context, i.e. hypotheses for fragments directly connected to one of the operands. So we introduce the **dependency** relation over fragments[6]. Suppose there is a symmetric dependency relation $D \subseteq F \times F$, where $F \subseteq 2^V$ is the set of all fragments in graph G. $D(x_i, x_j)$ means that any operation on a fragment hypothesis of x_i / x_j depends on the features in the fragment hypothesis of x_j / x_i. So relation D is symmetric.

We are especially interested in the following two dependency relations.

- *level-0 dependency*: $D_0(x_i, x_j)$ if and only if $i = j$.
- *level-1 dependency*: $D_1(x_i, x_j)$ if and only if x_i and x_j are directly connected in G.

5 Incremental parsing is only a special case of the incremental construction defi ned here. In general, one can search the graph in any order, including from left to right in a linear graph as in incremental parsing.

6 *Dependency* relation over fragments is different from the *dependency* in dependency parsing

So, in the incremental construction, we need to introduce a data structure to maintain the hypotheses with dependency relations among them.

A set of fragments, $c = \{x_1, x_2, ..., x_n\}$, is called a chain of fragments depending on x_i, $i = 1..n$, or **chain** for x_i for short, for a given symmetric dependency relation D, if

- For any two fragments x_i and x_j in c, $x_i \cap x_j = \emptyset$.
- For any two fragments x_i and x_j in c, there exists $x_{k,0}, ..., x_{k,m}$, such that $x_{k,0} = x_i, x_{k,m} = x_j$ and $D(x_{k,j-1}, x_{k,j})$.

For a given chain $c = \{x_1, x_2, ..., x_n\}$, we use $h^c = \{y^{x1}, ..., y^{xn}\}$ to represent a set of fragment hypotheses for the fragments in c, where y^{xi} is a fragment hypothesis for x_i in c. h^c is called a **chain hypothesis** for chain c. We use $H^c = \{h^c_1, ..., h^c_m\}$ to represent a set of chain hypotheses for chain c.

Now we can divide a given graph $G(V,E)$ with chains. A **cut** T of a given G, $T = \{c_1, c_2, ..., c_m\}$, is a set of chains satisfy

- exclusiveness: $\bigcup c_i \cap \bigcup c_j = \emptyset, \forall i, j$, and
- completeness: $\bigcup(\bigcup T) = V$.

Furthermore, we use $H^T = \{H^c | c \in T\}$ to represent of sets of chain hypotheses for all the chains in cut T. During the greedy search, we always maintain one cut over the whole graph. At the beginning, we have n chains, where n is the number of vertices. We merge chains step by step, and obtain a single chain for the whole graph.

As noted in the previous section, the idea behind the chain structure is that

- *hypotheses for the same chain are mutually exclusive*, and
- *hypotheses for different chains are compatible with each other*

In this way, we can generate hypotheses for different chains in parallel from different starting points in a graph. Two chains merge into one after an operation on both of them.

3.2 Search Algorithm

Algorithm 1 describes the procedure of building hypotheses incrementally on a given graph $G(V,E)$. Parameter k is used to set the beam width of search. Weight vector **w** is used to compute the score of an operation.

We first initiate the cut T by treating each vertex in V as a fragment and a chain. Then we set the initial hypotheses for each vertex/fragment/chain. For example, in dependency parsing, the initial value is a set of possible POS tags for each single word. Then we use a priority queue Q to collect all the possible operations over the initial cut T and hypotheses H^T.

Whenever Q is not empty, we search for the chain hypothesis with highest score on operation according to a given weight vector **w**. Suppose we find a new (fragment, hypothesis) pair (x,y) which is generated by the operation with the highest score. We first update the cut and the hypotheses according to (x,y). Let c^x be the chain for x. We remove from the cut T all the chains that overlap with c^x, and add c^x to T. Furthermore, we remove the chain hypotheses for those removed chains, and add the top k chain hypotheses for c^x to H^T. Then, we update the candidate queue Q by removing operations depending on the chain hypotheses that has been removed from H^T, and adding new operations depending on the chain hypotheses of c^x.

Algorithm 1 Incremental Construction

```
Require: graph $G(V,E)$;
Require: beam width $k$;
Require: weight vector $\mathbf{w}$;
1: cut $T \leftarrow initCut(V)$;
2: hypotheses $H^T \leftarrow initHypo(T)$;
3: candidate queue $Q \leftarrow initQueue(H^T)$;
4: repeat
5:     $(x',y') \leftarrow arg\max_{(x,y)\in Q} score(y)$;
6:     $T \leftarrow updCut(T,x',y')$;
7:     $H^T \leftarrow updHypo(H^T,x',y',k)$;
8:     $Q \leftarrow updQueue(Q,x',y',H^T)$;
9: until $(Q = \emptyset)$
```

Now we explain the functions in Algorithm 1 one by one.

- $initCut(V)$ initiates a cut T with vertices V by setting $T = \{c_i\}$, where $c_i = \{x_i\}$, and $x_i = \{v_i\}$ for each $v_i \in V$. This means that we take each vertex as a fragment, and each fragment constitutes a chain.
- $initHypo(T)$ initiates hypothesis H^T with the cut T described above. Here we set the initial fragment hypotheses, $Y^{xi} = \{y_1^{xi}, \ldots, y_m^{xi}\}$, where $x_i = \{v_i\}$ contains only one vertex, and $m \leq$ beam width k.
- $initQueue(H^T)$ initiates the queue of candidate operations over the current cut T and H^T. Supposed there exist v_i and v_j which are directly connected in G. Let

  $$C = \{c^{xi}, c^{xj}\} \cup N(D,x_i) \cup N(D,x_j),$$

  where $N(D,x_i) = \{c^x | D(x_i,x), c^x \in T\}$ is the set of chains one of whose fragments depends on x_i. For example, in Figure 1, let v_1 = *graduate*, v_2 = *students* and v_3 = *were*. We have

  $$N(D,x_2) = \{c^{x1}, c^{x3}\} = \{\{x_1\},\{x_3\}\},$$

  since we have $D(x_1,x_2)$ and $D(x_2,x_3)$ in the initial cut T. Similarly, we have $N(D,x_1) = \{\{x_2\}\}$. Therefore, if we consider the candidate operations that will merger $\{x_1\}$ and $\{x_2\}$, we have

  $$C = \{c^{x1}, c^{x2}\} \cup \{c^{x2}\} \cup \{c^{x1}, c^{x3}\} = \{\{x_1\},\{x_2\},\{x_3\}\}$$

  Then, we apply all possible operations to all compatible fragment hypotheses of x_i and x_j with respect to all possible chain hypotheses combinations for C, and put them in Q. In the example described above, we need to enumerate all the possible segment hypotheses combinations for x_1, x_2, x_3 with all possible operations over x_1 and x_2.

Suppose we generate (x_p, y^{xp}) with some operation, where x_p is equivalent to $x_i \cup x_j$, we have

$$c^{xp} = \bigcup C \cup \{x_p\} \setminus \{x_i, x_j\}.$$

In the previous example, we have $x_{1,2} = \{v_1, v_2\}$ and $c^{x1,2} = \{x_1, x_2, x_3\} \cup \{x_{1,2}\} \setminus \{x_1, x_2\} = \{x_{1,2}, x_3\} = \{\{v_1, v_2\}, \{v_3\}\}$. It should be noted that x_p is just a candidate here, and we do not update the cut T with respect to any candidate at this step.
All the candidate operations are organized with respect to the chain that each operation generates. For each chain c, we maintain the top k candidates according to the score of the chain hypotheses. Scores of operations, fragments and chains are calculated with formula (2), (3) and (4) respectively.

- $updCut(T, x, y)$ is used to update cut T with respect to the candidate operation that generates $y^x = R(x_i, x_j, y^{xi}, y^{xj})$. Let $C = \{c^{xi}, c^{xj}\} \cup N(D, x_i) \cup N(D, x_j)$ as described above. We remove all the chains in C from T, and add c^x to T.

$$T \leftarrow T \cup \{c^x\} \setminus C$$

Suppose, we choose to merge *graduate* and *students*, now we have

$$T = \{\{\{v_1, v_2\}, \{v_3\}\}, \{\{v_4\}\}, \{\{v_5\}\}, \ldots\}.$$

Then we obtain the cut as shown if Figure 3.

- $updHypo(H^T, x, y, k)$ is used to update hypothesis H^T. We remove from H^T all the chain hypotheses whose corresponding chain has been removed from T in $updCut(T, x, y)$. Furthermore, we add the top k chain hypotheses for chain c^x to H^T.
- $updQueue(Q, x, y, H^T)$ is designed to complete two tasks. First it removes from Q all the chain hypotheses which depend on one of the chains in C. Then it adds new candidate chain hypotheses depending on chain hypotheses of c^x in a way similar to the $initQueue(H^T)$ function. In Q, candidate chain hypotheses are organized with respect to the chains. For each chain c, we maintain the top k candidates according to the score of the chain hypotheses.

3.3 Learning Algorithm

In the previous section, we described a search algorithm for graph-based incremental construction for a given weight vector **w**. In Algorithm 2, we present a Perceptron like algorithm to obtain the weight vector from the training data.

For each given training sample (G_r, H_r), where H_r is the gold standard hidden structure of graph G_r, we first initiate cut T, hypotheses H^T and candidate queue Q by calling *initCut*, *initHypo* and *initQueue* as in Algorithm 1.

Then we use the gold standard H_r to guide the search. We select candidate (x', y') which has the highest operation score in Q. If y' is compatible with H_r, we update T, H^T and Q by calling *updCut*, *updHypo* and *updQueue* as in Algorithm 1. If y' is incompatible with H_r, we treat y' as a negative sample, and look for a positive sample $\tilde{y}$ in Q with $schPosi(Q, x')$.

Algorithm 2 Learning Algorithm

```
1:  $\mathbf{w} \leftarrow \mathbf{0}$;
2:  for (round $r = 0$; $r < R$; $r$++) do
3:      load graph $G_r(V, E)$, hidden structure $H_r$;
4:      initiate $T, H^T$ and $Q$;
5:      repeat
6:          $(x', y') \leftarrow arg\max_{(x,y) \in Q} score(y)$;
7:          if ($y'$ is compatible with $H_r$) then
8:              update $T, H^T$ and $Q$ with $(x', y')$;
9:          else
10:             $\tilde{y} \leftarrow schPosi(Q, x')$;
11:             $promote(\mathbf{w}, \tilde{y})$;
12:             $demote(\mathbf{w}, y')$;
13:             update $Q$ with $\mathbf{w}$;
14:         end if
15:     until ($Q = \emptyset$)
16: end for
```

If there exists a hypothesis $\tilde{y}^{x'}$ for fragment x' which is compatible with H_r, then *schPosi* returns $\tilde{y}^{x'}$. Otherwise *schPosi* returns the candidate hypothesis which is compatible with H_r and has the highest score of operation in Q.

Then we update the weight vector $\mathbf{w}$ with $\tilde{y}$ and y'. At the end, we update the candidate Q by using the new weights $\mathbf{w}$ to compute the score of operation.

We can use various methods to improve the performance of the Perceptron algorithm. In our implementation, we use Perceptron with margin in the training (Krauth and Mezard 1987). The margins are proportional to the loss of the hypotheses. Furthermore, we use voted Perceptron (Freund and Schapire 1999; Collins 2002) for inference.

4. LTAG Dependency Parsing

In this section, we will illustrate the details for the LTAG dependency parser which is not covered in the example in Section 2. In Section 4.1, we will describe how the hidden structure U in the algorithms is implemented for LTAG dependency parsing. In Section 4.2, we will illustrate the features used in our parser.

4.1 Incremental Construction

With the "incremental" construction algorithm described in the previous sections, we build the LTAG dependency tree incrementally. A hypothesis of a fragment is represented with a sub dependency tree. When the fragment hypotheses of two nearby fragments combine, the two sub dependency trees are combined into one.

4.1.1 Adjunction. It seems trivial to combine two partial dependency (derivation) trees with attachment. We can simply attach the root of tree A to some node on tree B which is *visible* to tree A. However, if adjunction is involved, the operation becomes a little bit complicated. An adjoined subtree may be *visible* from the other side of the dependency (derivation) tree. This is usually called *wrapping adjunction*.

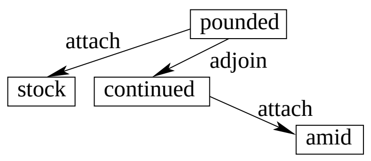

Figure 10
Wrapping adjunction with raising verbs

Wrapping adjunction may occur with passive ECM verbs as shown in Figure 9, or raising verbs as in the following example, shown in Figure 10.

Example 2
The stock of UAL Corp. continued to be pounded amid signs that British Airways ...
Here *continued* adjoins onto *pounded* [7], and *amid* attaches to *continued* from the other side of the dependency (derivation) tree (*pounded* is between *continued* and *amid*).

In order to implement wrapping adjunction, we introduce the idea of visibility. We will use a simple description and figures to define visibility. We define visibility with respect to the direction of operations. Without loss of generality, we only illustrate *visibility from right* here.

We define visibility recursively, starting from the root of a partial dependency tree. The root node is always visible. Assume that a node in a partial dependency tree is visible from right, or visible for short here. We search for the child nodes which are visible also. Let the parent node be N_T.

- If there is no adjunction from the left side, then the rightmost child, N_R, is visible, as in Figure 11.
- Otherwise, let N_L adjoin to N_T from the left side [8]. Let N_{LX} be the rightmost descendant in the sub tree rooted on N_L, and let N_{RX} be the rightmost descendant in the sub tree rooted on N_R.
  - If N_T is to the right of N_{LX}, then both N_L and N_R are visible, as in Figure 12.
  - If N_{LX} is to the right of N_T and N_{RX} is to the right of N_{LX}, then N_R is visible, as in Figure 13.
  - If N_{LX} is to the right of N_{RX}, then N_L is visible, as in Figure 14.

In this way, we obtain all the visible nodes recursively, and all the rest are invisible from right. If there are multiple adjunction from the left side, we need to compare among those adjoined nodes in a similar way, although this rarely happens in the real data.

4.1.2 Predicate Coordination. As we have noted, predicate coordination is represented explicitly in the LTAG-spinal Treebank. In order to build predicate coordination incrementally, we need to decompose coordination into a set of **conjoin** operations. Suppose a coordinated structure

7 In this case, the direction of LTAG dependency is opposite to the direction in traditional dependency analysis. This treatment makes the LTAG dependency more close to the deep structure. In addition, by expanding an LTAG dependency tree to a derivation trees, we could conveniently generate the an LTAG derived tree that represents the phrasal structures.

8 For the sake of ease in description, we assume there is only one adjunction from the left side. However, the reader can easily extend this to the case of multiple adjunctions.

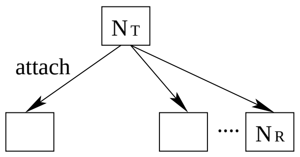

Figure 11
Case 1: no adjunction from left

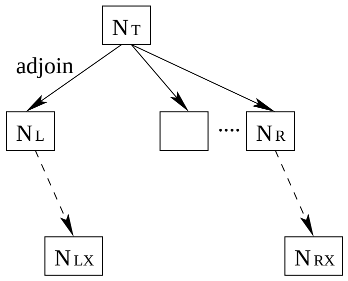

Figure 12
Case 2: both N_L and N_R are visible

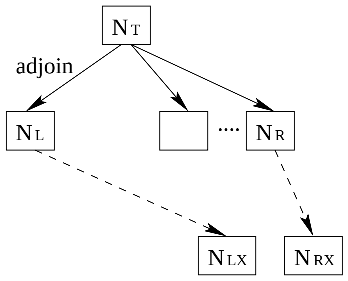

Figure 13
Case 3: N_R is visible

attaches to the parent node on the left side. We build this structure incrementally by attaching the first conjunct to the parent and conjoining other conjuncts to first one. In this way, we do not need to force the coordination to be built before the attachment. Either operation could be executed first.

Figure 15 shows the incremental construction of predicate coordination of the following sentence.

Example 3
I couldn't resist rearing up on my soggy loafers and saluting.

4.2 Features

In this section, we will describe the features used in LTAG dependency parsing. As to feature definition, an operation is represented by a 4-tuple

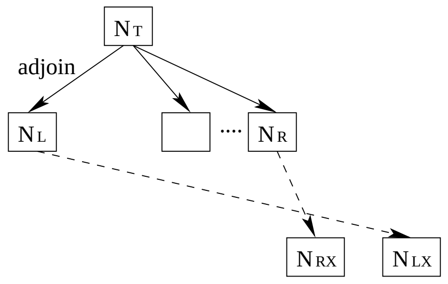

Figure 14
Case 4: N_L is visible

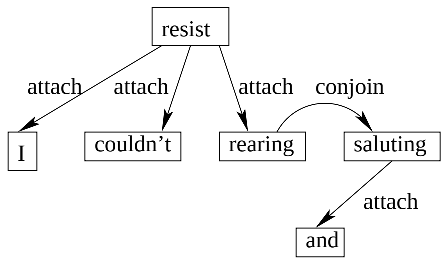

Figure 15
Partial dependency tree for the example of conjunction

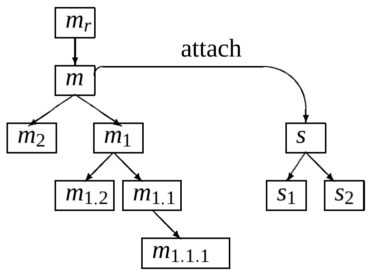

Figure 16
Representation of the nodes

- $op = (type, dir, pos_{left}, pos_{right})$,

where $type \in \{attach, adjoin, conjoin\}$ and dir is used to represent the direction of the operation. pos_{left} and pos_{right} are the POS tags of the two operands.

Features are defined on POS tags and lexical items of the nodes. In order to represent the features, we use m for the main-node of the operation, s for the sub-node, m_r for the parent of the main-node, $m_1..m_i$ for the children of m, and $s_1..s_j$ for the children of s. The index always starts from the side where the operation takes place. We use the Gorn address to represent the nodes in the subtrees rooted on m and s.

Table 1
Features defined on the context of operation

category	description	templates
one operand	Features defined on only one operand. For each template tp, $[type, dir, tp]$ is used as a feature.	$(m.p)$, $(m.w)$, $(m.p, m.w)$, $(s.p)$, $(s.w)$, $(s.p, s.w)$
two operands	Features defined on both operands. For each template tp, $[op, tp]$ is used as a feature. In addition, $[op]$ is also used as a feature.	$(m.w)$, $(c.w)$, $(m.w, c.w)$
siblings	Features defined on the children of the main nodes. For each template tp, $[op, tp]$, $[op, m.w, tp]$, $[op, m_r.p, tp]$ and $[op, m_r.p, m.w, tp]$ are used as features.	$(m_1.p)$, $(m_1.p, m_2.p)$, .., $(m_1.p, m_2.p, .., m_i.p)$
POS context	In the case that gold standard POS tags are used as input, features are defined on the POS tags of the context. For each template tp, $[op, tp]$ is used as a feature.	$(l_2.p)$, $(l_1.p)$, $(r_1.p)$, $(r_2.p)$, $(l_2.p, l_1.p)$, $(l_1.p, r_1.p)$, $(r_1.p, r_2.p)$
tree context	In the case that level-1 dependency is employed, features are defined on the trees in the context. For each template tp, $[op, tp]$ is used as a feature.	$(h_l.p)$, $(h_r.p)$
half check	Suppose $s_1, ..., s_k$ are all the children of s which are between s and m in the flat sentence. For each template tp, $[tp]$ is used as a feature.	$(s.p, s_1.p, s_2.p, .., s_k.p)$, $(m.p, s.p, s_1.p, s_2.p, .., s_k.p)$ and $(s.w, s.p, s_1.p, s_2.p, .., s_k.p)$, $(s.w, m.p, s.p, s_1.p, s_2.p, .., s_k.p)$ if $s.w$ is a verb
full check	Let x_1, x_2, .., x_k be the children of x, and x_r the parent of x. For any $x = m_{1.1...1}$ or $s_{1.1...1}$, template tp, $[tp(x)]$ is used as a feature.	$(x.p, x_1.p, x_2.p, .., x_k.p)$, $(x_r.p, x.p, x_1.p, x_2.p, .., x_k.p)$ and $(x.w, x.p, x_1.p, x_2.p, .., x_k.p)$, $(x.w, x_r.p, x.p, x_1.p, x_2.p, .., x_k.p)$ if $x.w$ is a verb

Furthermore, we use l_k and r_k to represent the nodes in the left and right context of the flat sentence. We use h_l and h_r to represent the head of the outside hypothesis trees on the left and right context respectively.

Let x be a node. We use $x.p$ to represent the POS tag of node x, and $x.w$ to represent the lexical item of node x.

Table 1 show the features used in LTAG dependency parsing. There are seven classes of features. The first three classes of features are those defined on only one operand, on both operands, and on the siblings respectively. If the gold standard POS tags are used as input, we define features on the POS tags in the context. If level-1 dependency is used, we define features on the root node of the hypothesis partial dependency trees in the neighborhood.

Half check features and full check features in Table 1 are designed for grammatical checks. For example, in Figure 16, node s attaches onto node m. Then nothing can attach onto s from the left side. The children of the left side of s are fixed, so we use the half check features to check the completeness of the children of the left half for s. Furthermore, we notice that all the left-edge descendants of s and the right-edge descendants of m become unavailable for any further operation. So their children are fixed after this operation. All these nodes are in the form of $m_{1.1...1}$ or $s_{1.1...1}$. We use full check features to check the children from both sides for these nodes.

5. Discussion

5.1 On Weight Update

Let us first have a close look at the function *schPosi*.

1. This function first tries to find a local correction to the wrong operation.
2. If it fails, which means that, in the gold standard, there is no direct operation over the two fragments involved, the function returns a correct operation which the current weight vector is the most confident of.

This learning strategy is designed to modify the current weight vector, or path preference, as little as possible, so that the context information learned previously will still be useful.

In case 1, only the weights of the features directly related to the mistake are updated, and unrelated features are kept unchanged.

In case 2, the feature vector of the operation with a higher score is closer to the direction of the weight vector. Therefore, to use the one with highest score helps to keep the weight vector in the previous direction.

5.2 On Path Selection

After the weight vector is updated, we re-compute the score of the candidate operations. The new operation with the highest score could still be over the same two fragments as the last round. However, it could be over other fragments as well. Intuitively, in this case, it means that the operation over the previous fragments is hard to decide with the current context, and we'd better work on other fragments first. This strategy is designed to learning a desirable order of operations.

Actually, the training algorithm makes the score of all operations comparable, since they always compete head to head in the queue of candidates. In this way, we can use the score to select the path.

5.3 Related Works

Yamada and Matsumoto (2003) has proposed a deterministic dependency parser which builds the parse tree incrementally via rounds of left-to-right scans. At each step, they check whether an attachment should be built or not. Each local decision is determined by a local classifier. This framework implements the bidirectional search mechanism via several rounds of scans.

Compared to their work, our parser has real bidirectional capability. At each step, we always compare all the possible operations. In addition, in our model, inference is incorporated in the training.

Variants of the Perceptron learning algorithms (Collins 2002; Collins and Roark 2004; Daumé III and Marcu 2005) have been successfully applied to left-to-right inference. However, it it not easy to directly use these algorithm for bidirectional learning, because we do not know the gold-standard direction of search. There are $O(n)$ options at the beginning, where n is the length of the input sentence. It does not sound good to update the weight vector with all these gold-standard compatible hypotheses as in (Daumé III and Marcu 2005), which conflicts with spirit with structure learning with Perceptron, *what is used for training is what we will meet in*

Table 2
Results of bidirectional dependency parsing on Section **22** of the LTAG-spinal Treebank, with different training and test beam widths. level-1 dependency.

test width	1	2	5	10
training width = 1	88.9	87.4	85.9	85.5
training width = 2	88.1	88.8	88.4	88.3
training width = 5	88.1	89.1	89.0	88.9
training width = 10	88.2	89.1	89.2	89.2

inference. We need to find out which gold-standard compatible hypothesis is more probable to be select in inference. Our learning algorithm has provided a very desirable solution.

In (Shen, Satta, and Joshi 2007), we reported a Perceptron learning algorithm for bidirectional sequence labeling, and achieved state-of-the-art performance on POS tagging. Our bidirectional parsing algorithm can be viewed as an extension of the labeling algorithm. However, we cannot fully utilize shared structure as in (Shen, Satta, and Joshi 2007) due to the difficulty of presenting non-projective dependency trees with simple states.

Similar to (Collins and Roark 2004) and (Daumé III and Marcu 2005), our training algorithm learns the inference in a subset of all possible contexts. However, our algorithm is more *aggressive*. In (Collins and Roark 2004), a search stops if there is no hypothesis compatible with the gold standard in the queue of candidates. In (Daumé III and Marcu 2005), the search is resumed after some gold-standard compatible hypotheses are inserted into a queue for future expansion, and the weights are updated correspondingly. However, there is no guarantee that the updated weights assign a higher score to those inserted gold-standard compatible hypotheses.

In our algorithm, the gold-standard compatible hypotheses are used for weight update only. As a result, after each sentence is processed, the weight vector can usually successfully predict the gold standard parse. As far as this aspect is concerned, our algorithm is similar to the MIRA algorithm in (Crammer and Singer 2003).

In MIRA, one always knows the correct hypothesis. However, in our case, we do not know the correct order of operations. So we have to use our form of weight update to implement aggressive learning.

In general, the learning model described in Algorithm 2 is more complex than supervised learning, because we do not know the correct order of operations. On the other hand, our algorithm is not as difficult as reinforcement learning, due to the fact that we can check the compatibility with the gold-standard for each candidate hypothesis.

The greedy search strategy used in inference algorithm for LTAG dependency parsing is similar to the Kruskal's minimal spanning tree algorithm (Kruskal 1956). However, in our case, the search space is limited. We do not consider all the possible dependency relations as in (McDonald et al. 2005). We only search the dependency structures that can be generated by LTAG derivation trees.

6. Experiments and Analysis

We use the same data set of the LTAG-spinal Treebank as in the left-to-right parser described in(Shen and Joshi 2005b). The LTAG-spinal Treebank (Shen and Joshi 2005a) was extracted from the Penn Treebank (PTB) with Propbank annotations. We train our LTAG dependency parser on section 2-21 of the LTAG-spinal Treebank. Section 22 is used as the development set for feature hunting. Section 23 is used for test.

Table 3
Results of bidirectional dependency parsing on Section **23** of the LTAG-spinal Treebank .

model	accuracy%
left-to-right, flex	89.3
level-0 dependency	90.3
level-1 dependency	90.5

We first study the effect of beam widths used in training and test. Table 2 shows the accuracy of dependency on the development data set with different beam settings on the level-1 model. The numbers show that larger beam width gives rise to better performance. However, the difference is little if we always use the same width for training and test. Even the deterministic model (beam width = 1) shows rather good performance.

Then we compare the level-0 model and the level-1 model, as shown in Table 3. Beam width is set to five for both training and test in both experiments. With level-0 dependency, our system achieves an accuracy of 90.3% at the speed of 4.25 sentences a second on a Xeon 3G Hz processor with JDK 1.5. With level-1 dependency, the parser achieves 90.5% at 3.59 sentences a second. Level-1 dependency does not provide much improvement due to the fact that level-0 features provide most of the useful information for this specific application. But this is not always true in other applications, e.g. POS tagging, in which level-1 features provide much more useful information.

It is interesting to compare our system with other dependency parsers. The accuracy on LTAG dependency is comparable to the numbers of the previous best systems on dependency extracted from PTB with Magerman's rules, for example, 90.3% in (Yamada and Matsumoto 2003) and 90.9% in (McDonald, Crammer, and Pereira 2005). However, their experiments are on the PTB, which is different from ours. To learn the LTAG dependency is more difficult for the following reasons.

Theoretically, the LTAG dependencies reveal deeper relations. Adjunction can lead to non-projective dependencies, and the dependencies defined on predicate adjunction are linguistically more motivated, as shown in the example in Figure 10. The explicit representation of predicate coordination also provides deeper relations. For example, in Figure 15, the LTAG dependency contains *resist* $\rightarrow$ *rearing* and *resist* $\rightarrow$ $saluting$, while the Magerman's dependency only contains *resist* $\rightarrow$ *rearing*. The explicit representation of predicate coordination helps to solve for the dependencies for shared arguments. So the LTAG dependencies reveal deeper relations, and is more difficult to learn at the syntax level.

For comparison, we also tried using Perceptron with *early stop* as described in (Collins and Roark 2004) to train the bi-directional parser. If the top hypothesis is incompatible with the gold-standard and there is no gold-standard compatible hypothesis for the same fragment in the beam, we stop training on this sentence. However, this method converges very slowly. After 20 iterations, the accuracy on the test data is 74.24%, and after 50 iterations, the accuracy increases to 75.96% only. Our learning algorithm takes less than 10 iterations to converge. It is also hard to apply the algorithm in (Daumé III and Marcu 2005) to our learning problem. As we have described before, there are $O(n)$ gold-standard compatible hypotheses at the beginning, where n is the length of the input sentence. Our aggressive learning algorithm clearly shows desirable performance for the learning of bi-directional inference.

7. Conclusions and Future Work

In this article, we have first introduced bidirectional incremental parsing, a new architecture of parsing for LTAG. We have proposed a novel algorithm for graph-based incremental construction, and applied this algorithm to LTAG dependency parsing, revealing deep relations, which are unavailable in other approaches and difficult to learn. We have evaluated the parser on the LTAG-spinal Treebank. Experimental results show a significant improvement over the incremental parser described in (Shen and Joshi 2005b). Graph-based incremental construction could be applied to other structure prediction problems in NLP.

As to the future work, we will explore in two different directions. We will extend our work to semantic role labeling and semantic parsing thanks to the extended domain of locality of LTAG. We are also interested in the methods of incorporating POS tagging and chunking into this discriminative learning framework. We hope to build an end-to-end system centering on LTAG derivation trees.

References

Abney, S. 1991. Parsing by chunks. In *Principle-Based Parsing*. Kluwer Academic Publishers.
Chiang, D. 2000. Statistical Parsing with an Automatically-Extracted Tree Adjoining Grammar. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL)*.
Collins, M. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference of Empirical Methods in Natural Language Processing*.
Collins, M. and B. Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*.
Crammer, K. and Y. Singer. 2003. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991.
Daumé III, H. and D. Marcu. 2005. Learning as search optimization: Approximate large margin methods for structured prediction. In *Proceedings of the 22nd International Conference on Machine Learning*.
Frank, R. 2002. *Phrase Structure Composition and Syntactic Dependencies*. The MIT Press.
Freund, Y. and R. E. Schapire. 1999. Large margin classifi cation using the perceptron algorithm. *Machine Learning*, 37(3):277–296.
Joshi, A. K. 1985. Tree adjoining grammars: How much context sensitivity is required to provide a reasonable structural description. In I. Karttunen, D. Dowty, and A. Zwicky, editors, *Natural Language Parsing*. Cambridge University Press, pages 206–250.
Joshi, A. K. and P. Hopely, 1997. *Extended Finite State Models of Language*, chapter A parser from Antiquity. Cambridge University Press.
Joshi, A. K. and Y. Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer-Verlag, pages 69 – 124.
Krauth, W. and M. Mezard. 1987. Learning algorithms with optimal stability in neural networks. *Journal of Physics A*, 20:745–752.
Kroch, A. and A. K. Joshi. 1985. The linguistic relevance of tree adjoining grammar. Report MS-CIS-85-16. CIS Department, University of Pennsylvania.
Kruskal, J. 1956. On the shortest spanning subtree and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50.
Lavelli, A. and G. Satta. 1991. Bidirectional parsing of lexicalized tree adjoining grammars. In *EACL 1991*.
Magerman, D. 1995. Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*.
Marcus, M. P., B. Santorini, and M. A. Marcinkiewicz. 1994. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.
McDonald, R., K. Crammer, and F. Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43th Annual Meeting of the Association for Computational Linguistics (ACL)*.
McDonald, R., F. Pereira, K. Ribarov, and J. Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*.

Palmer, M., D. Gildea, and P. Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1).

Rambow, O., D. Weir, and K. Vijay-Shanker. 2001. D-tree substitution grammars. *Computational Linguistics*, 27(1):89–121.

Sarkar, A. 2000. Practical experiments in parsing using tree adjoining grammars. In *Proceedings of the Fifth Workshop on Tree Adjoining Grammars and Related Formalisms: TAG+5*, Paris, France, May.

Sarkar, A. and A. K. Joshi. 1996. Coordination in tree adjoining grammars: Formalization and implementation. In *Proceedings of COLING '96: The 16th Int. Conf. on Computational Linguistics*.

Schabes, Y. and R. C. Waters. 1995. A cubic-time, parsable formalism that lexicalizes context-free grammar without changing the trees produced. *Computational Linguistics*, 21(4).

Schabes, Yves and Aravind K. Joshi. 1988. An earley-type parsing algorithm for tree adjoining grammars. In *COLING-ACL '98: Proceedings of 36th Annual Meeting of the Association for Computational Linguistics and 17th Int. Conf. on Computational Linguistics*.

Shen, L. and A. K. Joshi. 2005a. Building an LTAG Treebank. Technical Report MS-CIS-05-15, CIS Dept., Univ. of Pennsylvania. (submitted for publication).

Shen, L. and A. K. Joshi. 2005b. Incremental LTAG Parsing. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*.

Shen, L., G. Satta, and A. K. Joshi. 2007. Guided Learning for Bidirectional Sequence Classifi cation. In *(submitted for publication)*.

Toutanova, K., D. Klein, C. Manning, and Y. Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*.

van Noord, G. 1994. Head corner parsing for TAG. *Computational Intelligence*, 10(4).

Vijay-Shanker, K. 1987. *A study of Tree Adjoining Grammar*. Ph.D. thesis, University of Pennsylvania.

Vijay-Shanker, K. and A. K. Joshi. 1985. Some computational properties of tree adjoining grammars. In *ACL 1985*.

Yamada, H. and Y. Matsumoto. 2003. Statistical dependency analysis with Support Vector Machines. In *IWPT 2003*.

Zettlemoyer, L. and M. Collins. 2005. Learning to map sentences to logical form: Structured classifi cation with probabilistic categorial grammars.

Appendix

A. Lexicalized Tree Adjoining Grammar

In LTAG, each word is associated with a set of *elementary trees*. Each elementary tree represents a possible tree structure for the word [9].

There are two kinds of elementary trees, *initial trees* and *auxiliary trees*. Elementary trees can be combined through two operations, *substitution* and *adjunction*.

Substitution is used to attach an initial tree, and adjunction is used to attach an auxiliary tree. In addition to standard adjunction, we also use *sister adjunction* as defined in the statistical LTAG parser described in (Rambow, Weir, and Vijay-Shanker 2001; Chiang 2000) [10].

The tree resulting from the combination of elementary trees is is called a *derived tree*. The tree that records the history of how a derived tree is built from the elementary trees is called a *derivation tree* [11].

We illustrate the LTAG formalism with an example.

Example 4
Pierre Vinken will join the board as a non-executive director.

9 An elementary tree may have more than one lexical items.

10 Adjunction is used in the case where both the root node and the foot node appear in the parse tree. Sister adjunction is used in generating modifi er sub-trees as sisters to the head, e.g in base NPs.

11 Each node $\eta\langle n\rangle$ in the derivation tree is an elementary tree name η along with the location n in the parent elementary tree where η is inserted. The location n is the Gorn tree address (see Figure 20).

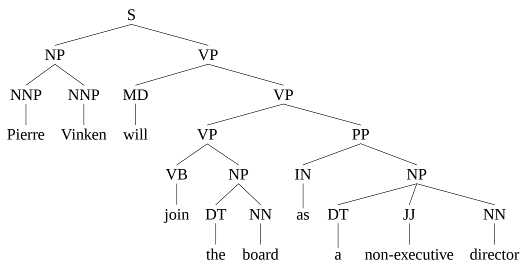

Figure 17
Derived tree (parse tree)

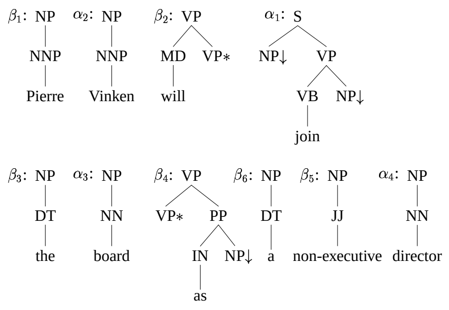

Figure 18
Elementary trees.

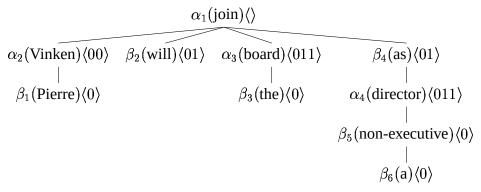

Figure 19
Derivation tree: shows how the elementary trees shown in Figure 18 can be combined to provide an analysis for the sentence.

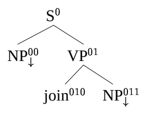

Figure 20
Example of how each node in an elementary tree has a unique node address using the Gorn notation. 0 is the root with daughters 00, 01, and so on recursively, e.g. ﬁrst daughter 01 is 010.

The derived tree for the example is shown in Figure 17. Figure 18 shows the elementary trees for each word in the sentence. Figure 19 is the derivation tree. α stands for an initial trees, and β stands for an auxiliary tree.

One of the properties of LTAG is that it factors recursion from the domain of dependencies, thus making all dependencies local in a sense. They can become long distance due to the adjunctions of auxiliary trees. For example, in the derivation tree, $\alpha_1(join)$ and $\alpha_2(Vinken)$ are directly connected no matter if there is an auxiliary tree $\beta_2(will)$ or not.

Compared with Context-Free Grammar (CFG), TAG is mildly context-sensitive (Joshi 1985), which means

- TAG can be parsed in polynomial time $O(n^6)$.
- TAG has constant growth property.
- TAG captures nested dependencies and limited kinds of crossing dependencies.

There exist non-context-free languages that can be generated by a TAG. For example, it can be shown that $L_4 = \{a^n b^n c^n d^n\}$ can be generated by a TAG, but it is not a context-free language. On the other hand, it can be shown that $L_5 = \{a^{n^2}\}$ is not a tree adjoining language (Vijay-Shanker 1987), but it is a context-sensitive language. It follows that $\mathcal{L}(CFG) \subset \mathcal{L}(TAG) \subset \mathcal{L}(CSG)$.

Because TAG has a stronger generative capacity (both week generative capability and strong generative capability) than CFG, we can use TAG to represent many structures that cannot be represented with CFG, mainly due to the adjunction operation. There follow the two key properties of LTAG:

- Extended Domain of Locality (EDL), which allows
- Factoring Recursion from the domain of Dependencies (FRD), thus making all dependencies local (Joshi and Schabes 1997).

These two properties reflect *the fundamental TAG hypothesis: Every syntactic dependency is expressed locally within a single elementary tree* (Frank 2002).

It is shown in (Kroch and Joshi 1985) and (Frank 2002) that a variety of constraints on transformational derivations can be nicely represented with TAG derivation without any stipulation on the TAG operations. This property is related to the fact that TAGs are not as strong as context sensitive grammars, which allow too much flexibility for natural language description.

B. LTAG-spinal

LTAG-spinal was introduced in (Shen and Joshi 2005a). It was mainly designed for statistical processing in the LTAG framework.

In LTAG-spinal, we have two kinds of elementary trees, initial trees and auxiliary trees, as shown in Figure 21. What makes LTAG-spinal different is that elementary trees are in the spinal form. A spinal initial tree is composed of a *lexical spine* from the root to the anchor, and nothing else. A spinal auxiliary tree is composed of a lexical spine and a *recursive spine* from the root to the foot node. The common part of a lexical spine and a recursive spine is called the *shared spine* of an auxiliary tree. For example, in Figure 21, the lexical spine for the auxiliary tree is $B_1, .., B_i, .., B_n$, the recursive spine is $B_1, .., B_i, .., B_1^*$, and the shared spine is $B_1, .., B_i$.

There are two operations in LTAG-spinal, which are *adjunction* and *attachment*. Adjunction in LTAG-spinal is the same as adjunction in the traditional LTAG. Attachment stems from *sister adjunction* in (Rambow, Weir, and Vijay-Shanker 2001; Chiang 2000). By attachment, we take the root of an initial tree as a child of a node of another spinal elementary tree.

An example of LTAG-spinal derivation tree is shown in Figure 22. In Figure 22, each arc is associated with a label which represents the type of operation. We use **A** for adjunction and **T** for attachment.

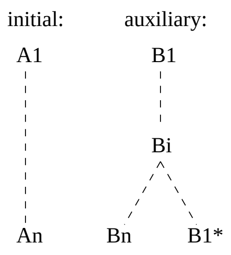

Figure 21
Spinal elementary trees

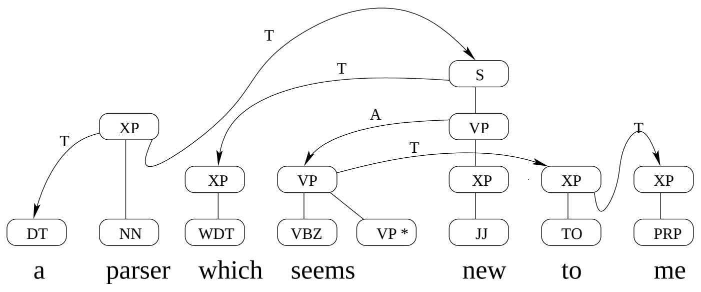

Figure 22
An example of LTAG-spinal derivation tree

It should be noted that the *adjunction* operation can effectively do wrapping, which distinguishes it from *sister adjunction*. In Figure 22, *seems* adjoins to *new* as a *wrapping adjunction*, which means that the leaf nodes of the adjunct subtree appear on both sides of the anchor of the main elementary tree in the resulting derived tree. Here, *seems* is to the left of *new* and *to me* is to the right of *new*. So the dependency relation for them is *non-projective*.

In (Shen and Joshi 2005a), an LTAG-spinal treebank was extracted from the Penn Treebank reconciled with Propbank annotation. Compared to the PTB, the LTAG-spinal treebank shows superior compatibility with the Propbank, which means it reveals deeper linguistic relation.