# A Survey on Left-to-Right Style Parsing Algorithms

Lingpeng Kong
lingpenk@cs.cmu.edu

November 26, 2013

**Abstract**

Parsing a sentence is a traditional task in the area of Natural Language Processing (NLP). Left-to-Right Style Parsing also has a very long history. Even before Dependency Parsing got a lot public attention, Left-to-Right Style Parsing algorithms were successfully applied in Phrase-Structure Parsing for quite a long time. This survey tries to summarize learning and inference algorithms used in Left-to-Right Style Parsing in both Phrase-Structure and Dependency Parsing, and attempts to seek a unified view of across the two.

## 1 Introduction

Left-to-right style parsing algorithms have quite a long history in Natural Language Processing, even in Computer Science, where one wants to parse computer programs written in a Context Free Grammar (CFG) (Chomsky, 1965). Left-to-right style parsing algorithms are preferred mainly for two reasons. First, it is very competitive in speed. Especially in dependency parsing, they achieve slightly lower than state-of-the-art performance in $O(n)$ runtime [1] (Nivre, 2004, 2009) where $n$ is the number of tokens in the sentence. Second, it corresponds to some appealing psycholinguistic theories which believe humans parse sentences from left to right (Marslen-Wilson, 1973; Frazier, 1987).

## 2 Phrase-Structure Parsing

Phrase-structure parsing lies at the center of Natural Language Processing techniques. Long before there was the Penn Treebank (Marcus et al., 1993), people have developed many algorithms to parse the natural language towards a phrase-structure tree (Earley, 1970; Kay, 1980; Tomita, 1987), where left-to-right style parsing plays an important role. After the construction of the treebank, left-to-right style parsing algorithms became one of the pioneers in introducing statistical methods into phrase-structure parsing (Magerman, 1995; Ratnaparkhi, 1997). Though people gradually moved their interests to PCFG, L-PCFG and re-ranking methods (Klein and Manning, 2003; Petrov et al., 2006; Charniak and Johnson, 2005) in the later days, we do find some attention is put back again to this old technique (Zhu et al., 2013), mainly motivated by the demands of speed.

### 2.1 Non-Statistical Phrase-Structure Parsers

The LR(k) parsing algorithms (Knuth, 1965) have been successfully applied to parse programming languages for a very long time. It is straightforward to think they can also be used in parsing natural languages. However, this adaptation has been proved to be difficult since the LR parsing algorithm can only handle a

[1] In non-projective dependency parsing, the expected runtime is linear but in worst case it is quadratic.

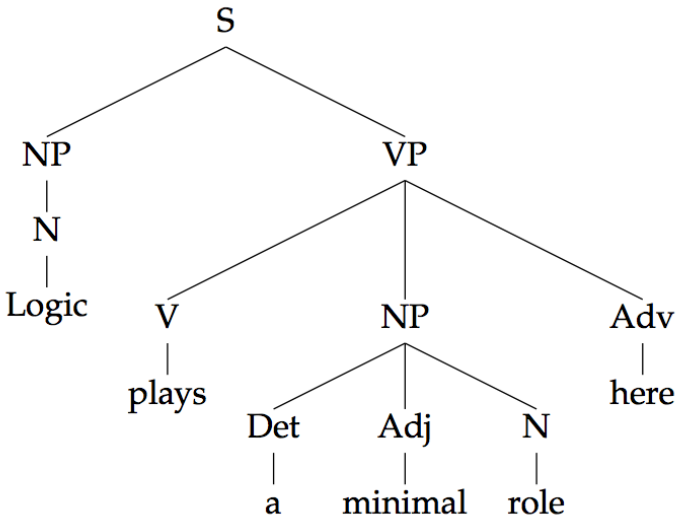


Figure 1: An example of a phrase-structure parsing tree from (Martins, 2012)

small subset of context free grammars called LR grammars, while natural language is surely beyond that. Typically when we use the LR algorithm to parse a sentence (or a statement in programming language), we will follow a LR parsing table constructed beforehand using only the grammar rules as inputs. The table will guide the parser to perform mainly two operations - to shift a token and then go to some next state, or to reduce and generate a non-terminal using some rule and go to some next state. If the grammar itself does not fall into the LR grammar category, as in the case of natural language, the algorithm will generate multiple operations in the same entry in the table. In the LR algorithm, when the parser encounters multiple entries, the parse fails, which is obviously not the thing we want in parsing natural language.

To solve the ambiguous grammar problem, Tomita (1987) proposed an efficient augmented-context-free parsing algorithm. By introducing the concept of "graph-structured stack", they managed to handle arbitrary ambiguous context-free grammars with at most polynomial space while preserving the speed advantage given by the LR algorithm. The idea of a "graph-structured stack" is quite intuitive. First, it worth noting that the multiple entries in the LR parsing table can be handled by a list of stacks, where a number of LR parsing processes run in a parallel manner. When a multiple entry occurs, we simply fork that process into two (or more), and each one will then perform one operation in the entry. By synchronizing the shift operation, we keep all the processes at the same pace. Then, by combining the stacks where the rightmost vertex is the same and processing them together and then combining the bottom portion of the stack (where the stack remains the same) together, they build the "graph-structured stack". Since they only let the ambiguous parts diverge in the graph while the same parts remain shared, they managed to parse the sentence efficiently.

Local ambiguity packing is used in the representation of the parsing forest to avoid the space growing exponentially with the ambiguities of the sentence. This is very much like what is usually see in the semiring parsing algorithms (Goodman, 1999), where inside one cell, the same non-terminal may have different derivations. In fact, Sikkel (1993) later showed that the Tomita parser can actually be represented inside the semiring framework.

Tomita's algorithm parses a sentence strictly from left to right, it enjoys quite good runtime and gives all paths analysis for the sentence. However, we cannot tell which parsing tree of the sentence is the most likely one (and we cannot only construct this one). Using a huge amount of CFG rules to construct a LR parsing table and then outputting the huge parsing forest based on the grammar is quite painful.

## 2.2 Statistical Phrase-Structure Parsers

### 2.2.1 Statistical Decision-Tree Model

Magerman (1995) developed a probability model for phrase-structure parsing. He defined the probability of a complete parse tree ($T$) of a sentence ($S$) to be the product of each decision($d_i$) conditioned on all previous decisions:

$$P(T|S) = \Pi_{d_i \in T} P(d_i | d_{i-1} d_{i-2} ... d_1 S) \quad (1)$$

The probability term $P(d_i|d_{i-1}d_{i-2}...d_1S)$ is actually modeled by three decision-tree models: a part-of-speech tagging model, a node-extension model, and a node-labeling model, each grown using questions on the word information and the context (both bigram, trigram context and constituent structures already built by the model – this is exactly where the $d_{i-1}d_{i-2}...d_1S$ terms come from).
The parsing algorithm is actually a search algorithm built on a stack decoding algorithm. First the machine runs a stack decoding algorithm to get a complete parse for the sentence with reasonable probability ($> 10^{-5}$). After that, a breadth-first search for all the partial parses which have not been explored by the stack decoder is performed. In this step, all the partial parse with a probability lower than the highest probability completed parse will be discarded.
In terms of learning, each state transition from $s_i$ to $s_{i+1}$ is used as a training example for the appropriate feature's tree, based on the event that happened in this transition (a tagging event for example), where the history consists of the answers to all the questions in state $s_i$. An adapted version of the CART algorithm (Breiman, 1993) is used to learn the decision tree from these training examples.

### 2.2.2 Statistical Parser Based on Maximum Entropy Models

Ratnaparkhi (1997) uses a maximum-entropy approach to build the incremental statistical parser with a observed running time O(n), where n is the length of the sentence. The general idea is very similar to the parser described by Magerman (1995). Essentially a history-based model, Ratnaparkhi (1997) uses the term "actions" while the Magerman (1995) uses the term "decisions". The score of a parse tree is defined as:

$$score(T) = \Pi_{a_i \in deriv(T)} q(a_i|b_i) \quad (2)$$

$b_i$ is the context in the moment of action i, which is the basically history as described in the work of Magerman (1995). A multi-pass algorithm is used to break the parsing into different phrases – tagging, NP chunking and build / checking. After the POS tagging and NP chunking, the parser start to uses the predicates (i.e. features) contains information in them to build the constituent in a BIO tagging fashion. The checking actions which are used to determine if the proposed constituent is completed or not, is quite similar to the shift-reduce decision in the Shift-Reduce Parser.
The three phrase linear model include different context information as features, but unlike SPATTER (Magerman, 1995), they merely contain words themselves rather than use some statistical clustering features which might need a typically expensive word clustering procedure. For example, the POS tagging model uses local features plus up-to second-order transitional features (features for $t_i$, $t_{i-1}$, $t_{i-2}$). The learning algorithm is Generalized Iterative Scaling (Darroch and Ratcliff, 1972) and in decoding they use basically beam-search, but plus a tag dictionary in the POS model stage to avoid heavy unnecessary computations.

## 2.3 Modern Shift-Reduce Constituent Parsers

Zhu et al. (2013) proposed a shift-reduce parser gives comparable accuracies to the state-of-art chart parsers like (Charniak and Johnson, 2005) and at the same time, the reported a speed-up from around 8 to 20 times then the chart parsers. The baseline parser they use is still a basic shift-reduce parser just like Ratnaparkhi (1997), where they define:

$$C(\alpha) = \sum_{i=1}^{N} \Phi(\alpha_i) \cdot \vec{\theta} \tag{3}$$

This is not much difference from the previous works but it is only a score (rather than a probability). By adding the scores for different actions together, it is just like the log probability thing defined in the previous section. They use the perceptron (Collins, 2002) algorithm to get the model parameter $\vec{\theta}$. The baseline feature they choose for this linear model is basically unigrams, bigrams, and trigrams in the stack and queue follow (Zhang and Clark, 2009). These give them 89.9% F-1 score in the English treebank.

A further 0.4 point is from the "padding" trick they use. Noticing the fact that the parse trees of the same

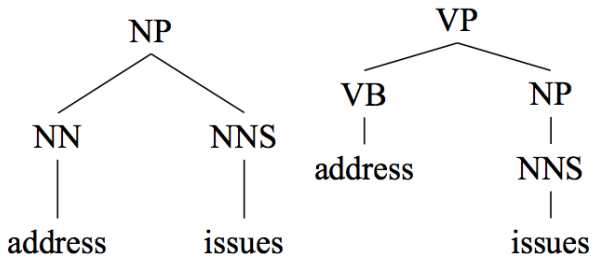


Figure 2: An example of parse trees of the same sentence with different numbers of actions from (Zhu et al., 2013)

sentence can have different number of actions (see Figure 2), which is mainly caused by the unary actions and can led to a gap of number of actions to $2n$ where $n$ is the number of words in the sentence, they introduce the "padding" trick to eliminate the impact led by this difference. For example, as we can see in Figure 2, before padding, the first parse takes {SHIFT, SHIFT, REDUCE-R-NP, FINISH} to build while the right one takes {SHIFT, SHIFT, UNARY-NP, REDUCE-L-VP, FINISH} to build – one more action than the left one. But after "padding" (i.e. introduce a new action called "IDLE" which does nothing but fill the gap) the left one becomes {SHIFT, SHIFT, REDUCE-R-NP, FINISH, IDLE} while the right one remains the same. The "IDLE" action will be decided by the model just like other actions.

Another characteristic which gives the parser around 1 point is the semi-supervise features. The semi-features they use including Brown Clustering features, lexical dependencies taken from dependency trees and dependency language model etc. However, these features slow down the whole parsing speed by 1/2. Although the parser itself is still much faster than those chart parsers.

# 3 Dependency Parsing

In this section, we review the use of left-to-right style parsing algorithms in the context of dependency parsing. First, we will introduce the definition of strict incremental dependency parsing and the reason why it is can not be done. Then, we will introduce the standard and arc-eager transition-based algorithms which lie at the core of the left-to-right style dependency parsing. The practical learning and inference algorithms for the transition-based systems will be discussed. After that, we review the potential drawbacks of this approach, and what can be done to solve or partially solve these problems (i.e. how to deal with non-projective case and how to reduce search error). Finally, the research attempting to bring together the graph-based and transition-based dependency parsers will be discussed.

## 3.1 Strict Incremental Dependency Parsing

Nivre (2004) gives a definition for strict incremental dependency parsing, which is, *at any point of during the parsing process, there is a single connected structure representing the analysis of the input consumed so far*. Say we are parsing the sentence from left to right, when the next word is $w_{i+1}$, $w_1$ to $w_i$ should form an connected graph. It is easy to show this can not be done in dependency parsing. In Figure 3, assuming we start to parse "the window", there is nothing we can do until we see the word "window", where the word "the" is disconnected from the previous words.

Despite the fact that strict incremental parsing can not be done in dependency parsing, the idea of shift-reduce algorithm can still be used here. In fact, the algorithm we will present in the following section (Table 1) is directly inspired by the classical shift-reduce algorithm where the "Left-Arc" and "Right-Arc" operations are very similar to "Reduce" operation in shift-reduce algorithm.

## 3.2 Transition-Based Dependency Parsing

### 3.2.1 Parsing Algorithm

In this section, we will first review the basic (standard) transition-based dependency parsing algorithm (Table 1). Although it is not really used in most modern transition-based dependency parsers, it is very helpful to start with that. Then we will explain the drawbacks of this algorithm and present the "arc-eager" algorithm (Table 2), which is the algorithm used in the popular transition-based dependency parser known as MaltParser (Nivre et al., 2006).

In this algorithm, the parser configuration is a triple $\langle S, I, A \rangle$, where $S$ is a of partially processed tokens, $I$

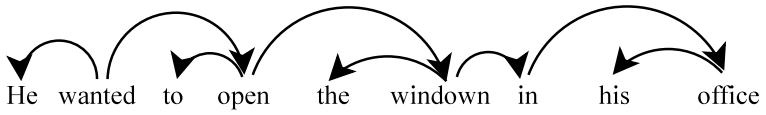


Figure 3: An Example Of A Dependency Tree

is a list for remaining input tokens, and $A$ is the current dependency arcs in the graph. The algorithm starts with a ROOT symbol in $S$ and the $I$ equals to the sequence of all the tokens in the sentence (since nothing has been parsed yet), and the set $A$ empty. We can do three operations.

**Shift** Shift a word $w_i$ on to $S$.

**Left-Arc** Build an arc from the front-most word $w_j$ of the list $I$, to the top word $w_i$ on the stack $S$, and pop the word $w_i$ to the stack.

| Action | | Precondition |
|---|---|---|
| Initialization | $\langle$ROOT, $W$, $\emptyset\rangle$ | |
| Termination | $\langle S$, nil, $A\rangle$ | |
| Left-Arc | $\langle w_i\|S, w_j\|I, A\rangle \rightarrow \langle S, w_j\|I, A \cup \{(w_j, w_i)\}\rangle$ | $\neg\exists w_k(w_k, w_i) \in A, w_i \neq$ ROOT |
| Right-Arc | $\langle w_i\|S, w_j\|I, A\rangle \rightarrow \langle S, w_i\|I, A \cup \{(w_i, w_j)\}\rangle$ | $\neg\exists w_k(w_k, w_j) \in A$ |
| Shift | $\langle S, w_i\|I, A\rangle \rightarrow \langle w_i\|S, I, A\rangle$ | |

Table 1: Basic Transition-Based Dependency Parsing

| Action | | Precondition |
|---|---|---|
| Initialization | $\langle$ROOT, $W$, $\emptyset\rangle$ | |
| Termination | $\langle S$, nil, $A\rangle$ | |
| Left-Arc | $\langle w_i\|S, w_j\|I, A\rangle \rightarrow \langle S, w_j\|I, A \cup \{(w_j, w_i)\}\rangle$ | $\neg\exists w_k(w_k, w_i) \in A, w_i \neq$ ROOT |
| Right-Arc | $\langle w_i\|S, w_j\|I, A\rangle \rightarrow \langle w_j\|w_i\|S, I, A \cup \{(w_i, w_j)\}\rangle$ | $\neg\exists w_k(w_k, w_j) \in A$ |
| Reduce | $\langle w_i\|S, I, A\rangle \rightarrow \langle S, I, A\rangle$ | $\exists w_j(w_j, w_i) \in A$ |
| Shift | $\langle S, w_i\|I, A\rangle \rightarrow \langle w_i\|S, I, A\rangle$ | |

Table 2: Arc-Eager Dependency Parsing

**Right-Arc** Build an arc from the top word $w_i$ on the stack $S$ to the front-most word $w_j$ on the list $I$, and **replace** $w_j$ with $w_i$ on $I$ then pop $w_i$.

This is actually very similar to the shift-reduce operation we have seen many times before, despite the fact that in reduce steps, rather than reduce the top two elements on the stack, here they do it for one element on the stack, the other on the input list. Note that some preconditions do exist. These preconditions help us to rule out invalid structures where ROOT has a parent or one token has two parent.

The problem with this algorithm is quite subtle. If we zoom into the **Right-Arc** step, we find that there is a important fact here, which is, before we can make an arc from $w_i$ to $w_j$, all the dependents of $w_j$ should all have been attached to $w_j$. To make this concrete, see the example in Figure 3. When we first see "wanted" at the top word in stack $S$ and "open" is the first word in list $I$, the algorithm can not attach "open" to "wanted" since after doing that, nothing can be attached to "open" anymore. Therefore, what the algorithm choose to do is keep doing *S*hift and *L*eft-Arc until the "office" and attached that to the word "in", and all the way back. This amount of shift operations in a continuous manner will cause too many unattached tokens in the stack, which is quite unfriendly to the machine learning algorithms introduced later.

To fix the problem, some small tricks need to be done on the standard transition-based dependency parsing algorithm, which leads to what we now know as "arc-eager" algorithm (Table 2). The central motivation for the arc-eager algorithm is, a head should be able to take a right dependent, before the dependents of the right dependent are found. To achieve this goal, the **Right-Arc** operation needs to be revised. Rather than **replace** $w_j$ with $w_i$ on $I$ then pop $w_i$, what the arc-eager algorithm does here, is to push $w_j$ on to the top of the stack so that $w_j$ can continue to hook up its dependent using the **Right-Arc** operation. Tokens that have gotten all their dependents in this way can leave the stack by applying the **Reduce** operation. Note that the precondition for the **Reduce** operation is that the word being reduced has a head node (which is obvious, the tokens which are reduced by this rule are introduced to the stack by **Right-Arc**, where the token must be a dependent node of someone).

To this point, there is a very important detail we miss in these algorithms, which is how to choose an operation to apply. The easiest way to make this decision is to just look at the the top word $w_i$ on the stack $S$ and

the first word $w_j$ in the list $I$, then decided if we want to build an arc from $w_i$ to $w_j$ (**Left-Arc**), to build an arc from $w_j$ to $w_i$ (**Right-Arc**), to shift $w_j$ to the stack $S$, or to reduce $w_i$ from the stack $S$ (**Reduce**). Of course the information in set $A$ will be used to check if the preconditions for the operation are met.
Although this is not exactly how modern transition-based parsers like MaltParser operate, but the basic intuition is the same. In reality (Nivre et al., 2006), a stack $C$ for context is also constructed to store the unattached tokens occurring between the token on the top of the stack $S$ and the next input token. A set of functions DEP, HEAD, LC, RC, LS, RS are used to check the head, dependent, leftmost child, rightmost child, left sibling and right sibling of the token. By building the system this way, transition-based dependency parsers can use a very rich set of features (including partially 2nd-order, 3rd-order feature, which is hard to access in graph-based models). By solving a classification problem using these features, the parser can parse the sentence smoothly.

### 3.2.2 Learning in Transition-Based Dependency Parsing

After we show the parsing algorithm for transition-based dependency parser in Table 1 and Table 2, the learning becomes straightforward. The center of the learning task for transition-based dependency parsers is to learn a classifier for actions based on the features mentioned on the previous sections. Many algorithms can be used here, while inside MaltParser, two models can be chosen from:

- Memory-based learning and classification (Daelemans and Van den Bosch, 2005), which basically store all the training instances and uses a KNN method to solve this multi-class classification problem.
- Support Vector Machine (Suykens and Vandewalle, 1999), which is a standard max-margin classifier where kernel methods are used to deal with non-linear boundary.

## 3.3 Non-Projective Transition-Based Dependency Parsing

As can be seen in the previous sections, both basic transition-based dependency parsing algorithm and arc-eager parsing algorithm are build inherently for projective dependency parsing. Non-projective parsing trees (see Figure 4) will not possible occurred in the results of the output of these algorithms. Nivre (2009) describes a SWAP trick (see Table 3) to handle the non-projective parsing trees. In Table 3, the notations are consistent with Table 1 and it is actually an added action for basic transition-based dependency parsing algorithm. In the SWAP action, we extract a word $w_j$ from (the second place of) the stack and put the word into the input list $I$. This is a very intuitive step, because by doing this, we can then build arcs that cross the arcs we have build using the standard actions in the table. One thing which should be noted here is that the prerequisite for this SWAP action is $j$ must less than $i$ (i.e. $w_j$ must occur before the word $w_i$ in the sentence).

By applying this trick, transitional-based parsers can effectively handle the non-projective case with a

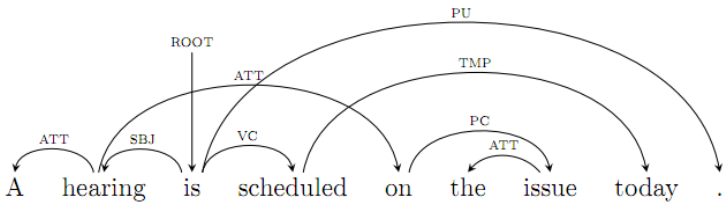


Figure 4: An Example of Non-projective Dependency Tree. A non-projective tree can have arcs go across each other.

| Action | | Precondition |
|---|---|---|
| SWAP | $\langle w_i, w_j \vert S, I, A\rangle \rightarrow \langle w_i \vert S, w_j \vert I, A\rangle$ | $i > j$ |

Table 3: SWAP Action in Non-Projective Dependency Parsing

worst-time complexity $O(n^2)$, but the expected time complexity is still linear, which means they still enjoys the fast paring time in reality.

## 3.4 Addressing Search Errors

### 3.4.1 Easy-First Non-Directional Dependency Parser

The main problem of the transition-based algorithm is that the decision is made greedily and extremely local. The algorithm can conditioned on very rich information from the left side (where all the parts have been parsed) while only a few words on the right side can be used. Easy-first non-directional dependency parsing algorithm (Goldberg and Elhadad, 2010) attempts to solve this problem by building the dependency tree in an iterative manner, where in each iteration, select the best pairs of neighbors to connect.

The parsing algorithm (see Figure 5) itself is very simple. First it scores the possible $2 \times (n-1)$ possible

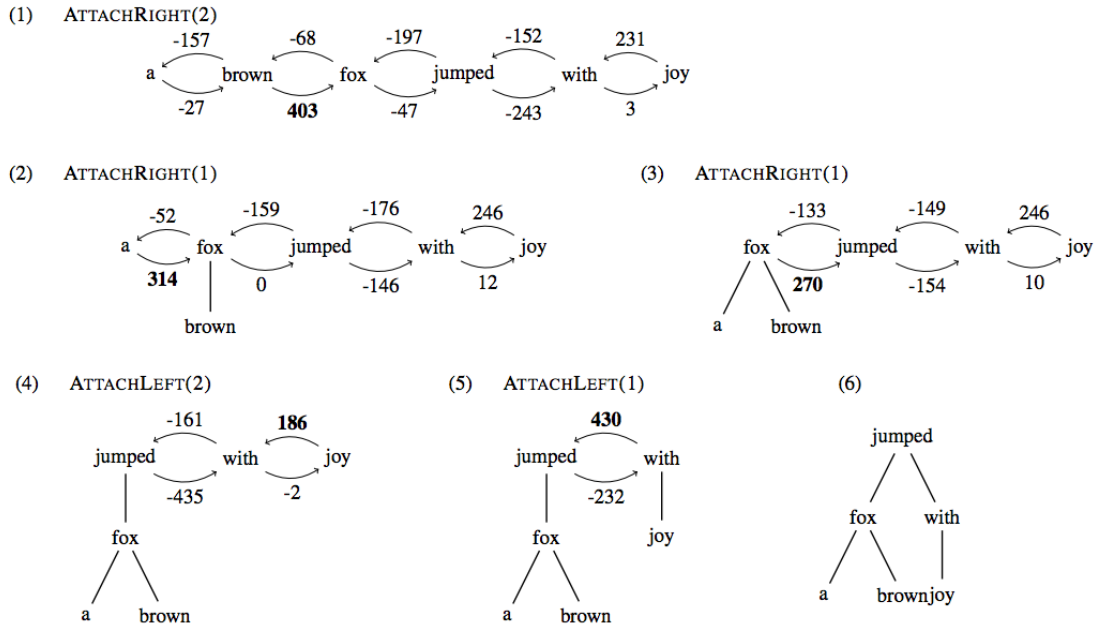


Figure 5: One run of the easy-first parsing algorithm from (Goldberg and Elhadad, 2010)

actions (build an arc from the $i$th part $p_i$ to the $(i+1)$th part $p_{i+1}$ or from $p_{i+1}$ to $p_i$) for a sentence with $n$ parts (at beginning, $n$ equals to the number of the words in the sentence). Then it performs the action with the highest score (which is the "easiest" one based on the algorithm's intuition). After that, there are $n-1$ parts left, and the algorithm repeats these steps until there is only one part left, which is the final projective dependency tree. The advantage of easy-first parsing algorithm is its ability to choose the "easiest" dependency arcs to build first, and leave the hard ones to be decided later, so that those decisions can be made on more information (more completed syntactical analysis of the sentence). Comparing to the left-to-right

transitional based system, easy-first can benefit more from the right side of the sentence while do not sacrifice the information from the left side.
Despite the simplicity of the easy-first parsing algorithm, easy-first parsing dose increase the complexity of the parsing algorithm comparing to the transitional-based parsers like MaltParser which typically takes $O(n)$ to parse one sentence (where $n$ is the length of the sentence) – Easy-first takes $O(n \log n)$ to parse. However, comparing to the graph-based dependency parsing systems, it still wins in algorithmic complexity. Graph-based dependency parsers generally take $O(n^2)$ (first-order) to $O(n^3)$ (second-order) to decode, since they do search the whole possible parsing trees and offer the global optimal solution.
The goal of learning in easy-first parsing algorithm is slightly different from the other parsers. It need to learn the weights in a linear model which score actions both correct and "easy" higher so that the parsing algorithm will choose them first. The learning algorithm is basically a variant of the structured perceptron (Collins, 2002). During the parsing, the learning algorithm will bring the weights of features associated with an error decision down while pull the weights of the current highest scored correct action up. Note that the correct action must meet two criteria. First, the generated arc must be inside the gold arcs. Second, the suggest child has found all its children (which is very similarity to the strategy in Arc-Standard Parsing). Parameter averaging is used to prevent the perceptron from over-fitting. In feature design, they design features like the POS-tags of left-most and right-most children of a word (which is an indicator of constituency) to signal the parser of incomplete phrase and PP-Attachment features to help learn lexicalized attachment preferences (the well-known pp-attachment ambiguous in parsing).
Just like the beam search technologies we will discussed in the following sections, easy-first parsing algorithm does not completely solve the searching error problem. The decisions made is still locally and greedily. But in terms of performance, it has two main contributions. First, it generally outperforms the transitional based parsing algorithms, although still under the graph-based parsing algorithms. Second, easy-first contribute to "parse diversity", where the parses produced by the non-directional parsers are different from both left-to-right transitional based and graph based parsing algorithms, which suggests combining the results of easy-first with the results produced by transitional and graph based systems might potentially leads to a performance gain.

### 3.4.2 From Greedy Search to Beam Search

Beam Search is very widely used in the world of transitional based dependency parsers (Zhang and Clark, 2008) mainly because their extremely local decisions may lead to serious problems known as cascade errors (i.e. if the parser makes errors in the previous decisions, those decisions will affect the later steps and make it highly possible to go wrong in the following decisions). Beam search is a simple but effective algorithm, the basic idea of the beam search is, rather than only keep the best one in each step, the search should expand the top k best decisions there, and pick the best one only in the last step. Although there is still no guarantee that this process will find the global optima in the whole solution space, it does give the algorithm more chance to avoid the error where the local best is beat by the decisions made in the later steps. The way which guarantee to find the global optima is dynamic programming, but in the world of context-free grammar parsing, dynamic programming can not be done in linear-time as Lee (2002) shows the CFG parsing can be used to compute matrix multiplication where a linear-time solution is far beyond reach.
Huang and Sagae (2010) propose a dynamic programming algorithm for shift-reduce parsing which runs polynomial time in theory but linear-time in practice. The algorithm is very related to (Tomita, 1987), and can be also regarded derived from the beam search algorithm. The two main points worth a note is 1) they merge the "equivalent states" in the same beam where the equivalent states means the states shifted to the same places and have an essentially same stack. Since they will have the same costs in the deductive system 2) they borrowed the idea of "graph-structured stack" from Tomita (1987) which effectively represent exponential number of trees in the parsing process. In terms of learning, they use averaged perceptron

(Collins, 2002) and "early updates" strategy. In both English and Chinese dependency parsing dataset, they both achieve very competitive results and the speed of the parser is fast.

### 3.4.3 Combining with Graph-Based Dependency Parsing

There is another line of research which seeks to combine the advantages of both transitional based dependency parsing and graph based dependency parsing. A very natural way to do that is to use stacking (Martins et al., 2008), where two level of parsers are built, level 0 from transitional-based parsing and level 1 uses the outputs of level 0 as features and is usually occupied by a graph-based dependency parser.
Zhang and Clark (2008) shows that beam-search is a competitive decoder for both graph-based and transitional based dependency parsers. They then use beam-search as a basis for combining the graph-based and transitional based parsers into one single system and achieve competitive results.

## 4 Conclusion

Left-to-right style parsing algorithms have been used for a long history in Natural Language Processing. The advantage of them are their very competitive speed (often linear in the length of the sentence). Since they formulate the decision problem as a multi-class classification problem, it is very straightforward for them to including very rich features which is harder to get in the chart / graph-based parsers due to the complicity in decoding. By applying a lot clever tricks, they can also meet different requirements (e.g. non-projective dependency parsing). However, they often suffer from the errors led by their greedy fashion. Even though they can represent the thing they are trying to optimize as a history-based scoring function, unlike the chart / graph-based algorithm, it is not very clear what they are trying to optimize represented by a function by the real inputs $X$ and real outputs $Y$. Both beam-search, dynamic programming, and other methods are used in addressing the search error problem, but there are still no general guarantee they can find the global optima. Ways to leverage the advantages of left-to-right style parsing are developed actively and a lot of potential remain to be researched.

## References


Breiman, L. (1993). *Classification and regression trees*. CRC press.

Charniak, E. and Johnson, M. (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 173–180. Association for Computational Linguistics.

Chomsky, N. (1965). Syntactic structures.

Collins, M. (2002). Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics.

Daelemans, W. and Van den Bosch, A. (2005). *Memory-based language processing*. Cambridge University Press.

Darroch, J. N. and Ratcliff, D. (1972). Generalized iterative scaling for log-linear models. *The annals of mathematical statistics*, 43(5):1470–1480.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Frazier, L. (1987). Syntactic processing: evidence from dutch. *Natural Language & Linguistic Theory*, 5(4):519–559.

Goldberg, Y. and Elhadad, M. (2010). An efficient algorithm for easy-first non-directional dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 742–750. Association for Computational Linguistics.

Goodman, J. (1999). Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Huang, L. and Sagae, K. (2010). Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086. Association for Computational Linguistics.

Kay, M. (1980). Algorithm schemata and data structures in syntactic processing. *Technical Report CSL80-12*.

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and control*, 8(6):607–639.

Lee, L. (2002). Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)*, 49(1):1–15.

Magerman, D. M. (1995). Statistical decision-tree models for parsing. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 276–283. Association for Computational Linguistics.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.

Marslen-Wilson, W. (1973). Linguistic structure and speech shadowing at very short latencies. *Nature*.

Martins, A. F., Das, D., Smith, N. A., and Xing, E. P. (2008). Stacking dependency parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 157–166. Association for Computational Linguistics.

Martins, A. F. T. (2012). *The Geometry of Constrained Structured Prediction: Applications to Inference and Learning of Natural Language Syntax*. PhD thesis, Columbia University.

Nivre, J. (2004). Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics.

Nivre, J. (2009). Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 351–359. Association for Computational Linguistics.

Nivre, J., Hall, J., and Nilsson, J. (2006). Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6, pages 2216–2219.

Petrov, S., Barrett, L., Thibaux, R., and Klein, D. (2006). Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics.

Ratnaparkhi, A. (1997). A linear observed time statistical parser based on maximum entropy models. *arXiv preprint cmp-lg/9706014*.

Sikkel, K. (1993). Parsing schemata.

Suykens, J. A. and Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300.

Tomita, M. (1987). An efficient augmented-context-free parsing algorithm. *Computational linguistics*, 13(1-2):31–46.

Zhang, Y. and Clark, S. (2008). A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 562–571. Association for Computational Linguistics.

Zhang, Y. and Clark, S. (2009). Transition-based parsing of the chinese treebank using a global discriminative model. In *Proceedings of the 11th International Conference on Parsing Technologies*, pages 162–171. Association for Computational Linguistics.

Zhu, M., Zhang, Y., Chen, W., Zhang, M., and Zhu, J. (2013). Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.