DEPENDENCY PARSING

JOAKIM NIVRE

Contents

A dependency parser analyzes syntactic structure by identifying dependency relations between words. In this lecture, I will introduce dependency-based syntactic representations (§1), arc-factored models for dependency parsing (§2), and online learning algorithms for such models (§3). I will then discuss two important parsing algorithms for these models: Eisner's algorithm for projective dependency parsing (§4) and the Chu-Liu-Edmonds spanning tree algorithm for non-projective dependency parsing (§5).

1. Dependency Trees

In a *dependency tree*, a sentence is analyzed by connecting words by binary asymmetrical relations called dependencies, which are categorized according to the functional role of the dependent word. Formally speaking, a dependency tree for a sentence x can be defined as a labeled directed graph $G = (V_x, A)$, where $V_x = \{0, \ldots, n\}$ is a set of nodes, one for each position of a word x_i in the sentence plus a node 0 corresponding to a dummy word ROOT at the beginning of the sentence, and where $A \subseteq (V_x \times L \times V_x)$ is a set of labeled arcs of the form (i, l, j), where i and j are nodes and l is a label taken from some inventory L. Figure 1 shows a typical dependency tree for an English sentence with a dummy root node.

Date: 2013-03-01.

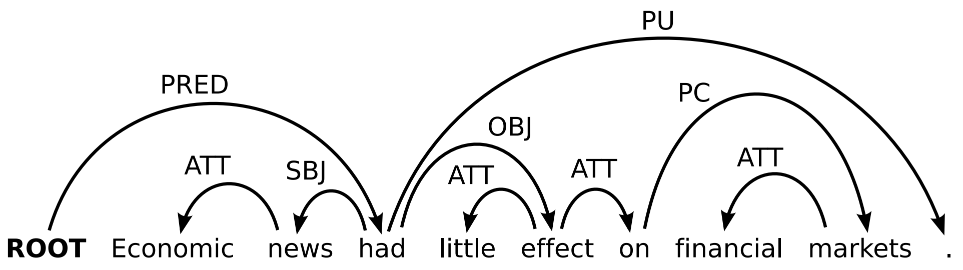

FIGURE 1. Dependency tree for an English sentence with dummy root node.

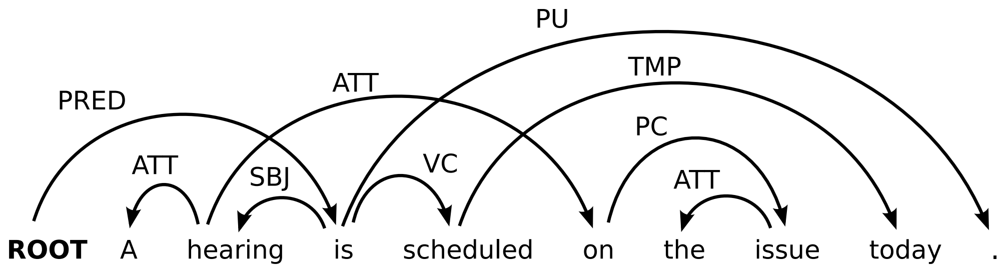

FIGURE 2. Non-projective dependency tree for an English sentence.

In order to be a well-formed dependency tree, the directed graph must also satisfy the following conditions:

(1) ROOT: The dummy root node 0 does not have any incoming arc (that is, there is no arc of the form $(i, l, 0)$).
(2) SINGLE-HEAD: Every node has at most one incoming arc (that is, the arc (i, l, j) rules out all arcs of the form (k, l', j) where $k \neq i$ or $l' \neq l$).
(3) CONNECTED: The graph is weakly connected (that is, in the corresponding undirected graph there is a path between any two nodes i and j).

In addition, a dependency tree may or may not satisfy the following condition:

(4) PROJECTIVE: For every arc in the tree, there is a directed path from the head of the arc to all words occurring between the head and the dependent (that is, the arc (i, l, j) implies that $i \rightarrow^* k$ for every k such that $\min(i, j) < k < \max(i, j)$).

Projectivity is a notion that has been widely discussed in the literature on dependency grammar and dependency parsing. Broadly speaking, dependency-based grammar theories and annotation schemes normally do not assume that all dependency trees are projective, because some linguistic phenomena involving discontinuous structures can only be adequately represented using non-projective trees. By contrast, many dependency-based syntactic parsers assume that dependency trees are projective, because it makes the parsing problem considerably less complex. Figure 2 shows a non-projective dependency tree for an English sentence.

The parsing problem for a dependency parser is to find the optimal dependency tree y given an input sentence x. Note that this amounts to assigning a syntactic head i and a label l to every node j corresponding to a word x_j in such a way that the resulting graph is a tree rooted at the node 0. This makes the parsing problem more constrained than in the case of phrase structure parsing, as the nodes are given by the input and only the arcs have to be inferred. In graph-theoretic terms, this is equivalent to finding a *spanning tree* in the complete graph $G_x = (V_x, V_x \times L \times V_x)$ containing all possible arcs (i, l, j) (for nodes i, j and labels l), a fact that is exploited in so-called graph-based models for dependency parsing.

Another difference compared to phrase structure parsing is that there are no part-of-speech tags in the syntactic representations (because there are no pre-terminal nodes, only terminal nodes). However, most dependency parsers instead assume that part-of-speech tags are part of the input, so that the input sentence x actually consists of tokens $x_1, \ldots, x_n$ annotated with their parts of speech $t_1, \ldots, t_n$ (and possibly additional information such as lemmas and morphosyntactic features). This information can therefore be exploited in the feature representations used to select the optimal parse, which turns out to be of crucial importance.

2. Arc-Factored Models

A widely used graph-based model for dependency parsing is the so-called arc-factored (or edge-factored) model, where the score of a dependency tree decomposes into the scores of individual arcs:[1]

$$\text{Score}(x, y) = \sum_{(i,l,j) \in A_y} \text{Score}(i, l, j, x) \tag{1}$$

We use A_y for the arc set of the dependency tree y (that is, $y = (V_x, A_y)$) and $\text{Score}(i, l, j, x)$ for the score of the arc (i, l, j) for the sentence x. This gives us a parsing model, where the generative component maps each sentence x to the set of all spanning trees for the node set V_x (relative to some label set L):

$$\text{GEN}(x) = \{y \,|\, y \text{ is a spanning tree in } G_x = (V_x, V_x \times L \times V_x)\} \tag{2}$$

The evaluative component then ranks candidate trees according to their arc-factored score and returns the one with the highest score:

$$y^* = \operatorname*{argmax}_{y \in \text{GEN}(x)} \text{EVAL}(x, y) = \operatorname*{argmax}_{y \in \text{GEN}(x)} \sum_{(i,l,j) \in A_y} \text{Score}(i, l, j, x) \tag{3}$$

The scoring function for arcs can be implemented in many ways but usually takes the form of a linear model:

$$\text{Score}(i, l, j, x) = \sum_{k=1}^{K} \mathbf{f}_k(i, l, j, x) \cdot \mathbf{w}_k \tag{4}$$

In this model, every function $\mathbf{f}_k(i, l, j, x)$ is a numerical (often binary) feature function, representing some salient property of the arc (i, l, j) in the context of x, and $\mathbf{w}_k$ is a real-valued feature weight, reflecting the tendency of $\mathbf{f}_i(i, l, j, x)$ to co-occur with good or bad parses. Note that this is a linear model of essentially the same type that we saw in the previous lecture, but it is not a log-linear model because the scores are not normalized to form a probability distribution. As a matter of historical tradition, dependency parsers tend to use purely discriminative models instead of probability models, but there is no principled reason why this should be the case. With a linear discriminative scoring model for arcs, the final form of the arc-factored model becomes:

$$y^* = \operatorname*{argmax}_{y \in \text{GEN}(x)} \text{EVAL}(x, y) = \operatorname*{argmax}_{y \in \text{GEN}(x)} \sum_{(i,l,j) \in A_y} \sum_{k=1}^{K} \mathbf{f}_k(i, l, j, x) \cdot \mathbf{w}_k \tag{5}$$

Since this model contains no grammatical rules at all, parsing accuracy is completely dependent on the choice of good features. Table 1 shows a selection of typical feature templates for arc-factored dependency parsing.

3. Online Learning

Learning the weights of an arc-factored model can be done in many ways, but the most popular approach is to use an online learning algorithm that parses one sentence at a time and updates the weights after each training example with the goal of minimizing the number of errors on the training corpus. Here we will only consider the structured perceptron algorithm, which is the simplest of these learning algorithms. For a discussion of more sophisticated margin-based methods, see McDonald et al. (2005a).

[1] This model is sometimes referred to as a first-order graph-based model, because it only considers one arc in isolation. In the next lecture, we will consider higher-order graph-based models, which incorporate structures consisting of more than one arc.

Unigram Features	**Bigram Features**	**In-Between PoS Features**
x_i-word, x_i-pos	x_i-word, x_i-pos, x_j-word, x_j-pos	x_i-pos, b-pos, x_j-pos
x_i-word	x_i-pos, x_j-word, x_j-pos	**Surrounding PoS Features**
x_i-pos	x_i-word, x_j-word, x_j-pos	x_i-pos, x_{i+1}-pos, x_{j-1}-pos, x_j-pos
x_j-word, x_j-pos	x_i-word, x_i-pos, x_j-pos	x_{i-1}-pos, x_i-pos, x_{j-1}-pos, x_j-pos
x_j-word	x_i-word, x_i-pos, x_j-word	x_i-pos, x_{i+1}-pos, x_j-pos, x_{j+1}-pos
x_j-pos	x_i-word, x_j-word	x_{i-1}-pos, x_i-pos, x_j-pos, x_{j+1}-pos
	x_i-pos, x_j-pos	

TABLE 1. Arc-factored feature templates $\mathbf{f}_k(i, l, j, x)$ for the arc (i, l, j) in the context of sentence x (McDonald, 2006). Notation: w-word = word form of w; w-pos = part-of-speech tag of w; b = any word between w_i and w_j. Binarization is used to obtain one binary feature for each possible instantiation of a template.

The structured perceptron algorithm was proposed by Collins (2002) as a generalization of the basic perceptron algorithm (Rosenblatt, 1958). The general idea in both cases is that we start with some initial weights (usually zero) and then process each training example using our current weights. If our model predicts the correct output, we do nothing (because our model did not perform an error). If our model predicts an incorrect output, we update our weights in such a way that we are less likely to make an error on the same example next time, by simply adding weight to the features of the correct output and subtracting weight from the features of the incorrectly predicted output. We keep iterating over the training examples multiple times either until we reach some termination criterion or (more commonly) for a fixed number of iterations.

The structured perceptron algorithm, as it applies to the arc-factored model for dependency parsing, is given in Figure 3. After the entire weight vector $\mathbf{w}$ has been initialized to zero (line 1), there is an outer loop over training iterations (line 2) and an inner loop over sentences in the training set (line 3). For each sentence on each iteration, we parse the sentence using the current weights (line 4). If the output parse y^* is different from the treebank parse y^i for sentence x^i (line 5), we update the weight vector with respect to y^* and y^i (line 6). After we have completed all iterations, we return the final weight vector (line 7).[2]

The learning algorithm in Figure 3 uses two subroutines. The subroutine $\textsc{Parse}(x^i, \mathbf{w})$ is simply a parser, or decoder, for the arc-factored model, which is the topic of the next two sections and will be treated as a black box for now. The subroutine $\textsc{Update}(\mathbf{w}, y^*, y^i)$ is specified in Figure 3 and updates the weight $\mathbf{w}_k$ by *subtracting* the value $\mathbf{f}_k(i, l, j, x)$ for every arc (i, l, j) in the predicted parse y^* and *adding* the value $\mathbf{f}_k(i, l, j, x)$ for every arc (i, l, j) in the predicted parse y^*. Assuming that all features are binary, this will be equivalent to updating $\mathbf{w}_k$ by the difference between the number of times the feature $\mathbf{f}_k(i, l, j, x)$ is active (has value 1) in the good and bad parse, respectively. Other things being equal, this means that the good parse will get a higher score next time, while the bad parse will get a lower score. Whether the model will actually prefer the good parse next time depends on the magnitude of the difference as well as what happens with other sentences in the training set until the next iteration.

4. Eisner's Algorithm

In order to use the arc-factored model for parsing, we need an efficient algorithm for decoding, that is, for finding the highest scoring dependency tree y for a given sentence x. Moreover, if we

[2]In practice, better results are usually obtained if, instead of returning the final weight vector, we return the average of all weight vectors seen during training. For a theoretical motivation of averaging, see Collins (2002).

```
Training data: $\mathcal{T} = \{(x^i, y^i)\}_{i=1}^{|\mathcal{T}|}$
1 $\mathbf{w} \leftarrow 0$
2 for $n : 1..N$
3     for $i : 1..|\mathcal{T}|$
4         $y^* \leftarrow$ PARSE$(x^i, \mathbf{w})$
5         if $y^* \neq y^i$
6             $\mathbf{w} \leftarrow$ UPDATE$(\mathbf{w}, y^*, y^i)$
7 return w

PARSE$(x, \mathbf{w})$
1 return $\text{argmax}_{y \in \text{GEN}(x^i)} \sum_{(i,l,j) \in A_y} \sum_{k=1}^{K} \mathbf{f}_k(i, l, j, x^i) \cdot \mathbf{w}_k$

UPDATE$(\mathbf{w}, y^*, y^i)$
1 for $k : 1..K$
2     for $(i, l, j) \in A_{y^*}$
3         $\mathbf{w}_k \leftarrow \mathbf{w}_k - \mathbf{f}_k(i, l, j, x)$
4     for $(i, l, j) \in A_{y^i}$
5         $\mathbf{w}_k \leftarrow \mathbf{w}_k + \mathbf{f}_k(i, l, j, x)$
```

FIGURE 3. The structured perceptron algorithm for arc-factored dependency parsing.

want to use perceptron learning (or similar algorithms), an efficient decoding algorithm is needed also for learning the weights of the model. In this section, we look at the standard algorithm for arc-factored dependency parsing under the projectivity constraint: Eisner's algorithm (Eisner, 1996, 2000). In the next section, we consider the more general case, where non-projective trees are allowed as well.

As long as we restrict ourselves to projective trees, we can build dependency trees by a dynamic programming algorithm very similar to the CKY algorithm. In fact, we can define a special kind of head-lexicalized (P)CFG that generates parse trees isomorphic to dependency trees. However, besides the fact that the arc-factored model does not have a any proper grammar rules, the time complexity of lexicalized PCFG parsing is $O(n^5)$, while Eisner's algorithm achieves lexicalized parsing in $O(n^3)$ by exploiting the special properties of dependency trees. The key idea is to use a split-head representation, that is, to let the chart cells represent half-trees instead of trees, which means that we can replace the indices for lexical heads (one in each half of the new subtree) by boolean variables indicating whether the lexical head is at the left or at the right periphery of the respective half-trees. In addition, we need to two combinatory operations, one for adding a dependency arc between the heads of two half-trees to form an incomplete half-tree, and one for subsequently combining this incomplete half-tree and with a complete half-tree to form a larger complete half-tree. Figure 4 gives a schematic representation of this process for left-headed dependencies and contrasts it with the CKY algorithm.

Pseudo-code for Eisner's algorithm can be found in Figure 5. First of all, note that this is an algorithm for unlabeled parsing, that is, where an arc is simply a pair of nodes (i, j) without any label. This is because, in the arc-factored model, the highest scoring labeled dependency tree is always the highest scoring unlabeled dependency tree with the highest-scoring label for each arc (because there is no interaction between the choice of labels for different arcs). Therefore, we can improve efficiency during parsing by only adding labels in a post-processing step, possibly using a different model for choosing labels. Eisner's algorithm is a dynamic programming algorithm

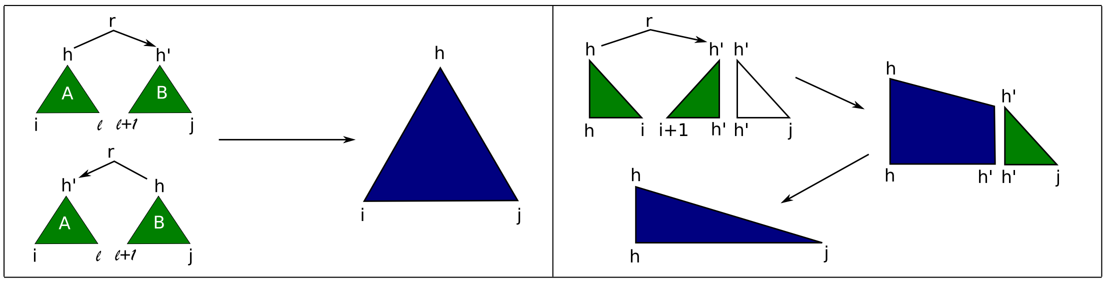

FIGURE 4. Comparison of the chart items in the CKY algorithm (left) and Eisner's algorithm (right).

that computes chart items of the form $C[i][j][d][c]$, where i and j are the leftmost and rightmost words of the substring spanned by the item, d is $\leftarrow$ if the head word is at the right periphery and $\rightarrow$ if it is at the left periphery, and c is 1 for complete items – a word and all its left or right half-tree – and 0 for incomplete items – two words connected by an arc and their inside half-trees (cf. Figure 4). The value stored in $C[i][j][d][c]$ is the score of the highest scoring substructure with this signature. To be able to retrieve the actual structure we need to maintain a corresponding chart with back-pointers, which are omitted in Figure 5.

Parsing is initialized by setting the score of every possible one-word item (left-headed, right-headed, complete and incomplete) to zero (lines 1–2). We then loop over span lengths m from smaller to larger (line 3) and over start positions i from left to right (line 4), setting the end position to $i + m$ (line 5). We first find the best incomplete items from i to j by finding the best split position k and adding up the scores of the left and right halves plus the score of the new arc, either (j, i) (line 6) or (i, j) (line 7). Note that the two (complete) items being combined must have i and j as their heads. Next, we find the best complete items from i to j by again finding the best split position k but this time only adding up the scores of one incomplete and one complete item having the same head, producing a complete item with the head to the right (line 8) or to the left (line 9). Finally, we return the highest-scoring complete item spanning 0 to n and with its head to the left.[3] It is easy to see that the parsing complexity is $O(n^3)$ since there are two nested **for** loops plus a loop hidden in the max operation, all of which take $O(n)$ time. This is considerably better than $O(n^5)$ for a naive adaptation of the CKY algorithm.

Eisner's algorithm was first put to use in Eisner (1996), who combined it with a generative probability model to create one of the first statistical dependency parsers. Later on, it was picked up by McDonald et al. (2005a), who instead used an arc-factored discriminative model and online learning (as described in the two preceding sections) to reach state-of-the-art accuracy for dependency parsing of English and Czech.

5. Spanning Tree Parsing

We noted earlier that a dependency tree y for a sentence x is a (directed) spanning tree in the complete graph $G_x = (V_x, V_x \times L \times V_x)$ (or $G_x = (V_x, V_x \times V_x)$ for unlabeled trees). Moreover, given an arc-factored model, finding the highest scoring dependency tree y for a sentence x is equivalent to the graph-theoretic problem of finding a weighted maximum spanning tree in G_x, where the weight of an arc (i, l, j) is simply $\text{SCORE}(i, l, j, x)$. From this perspective, Eisner's algorithm can be regarded as an algorithm for finding a *projective* maximum spanning tree. So

[3]This presupposes that we use a dummy root node indexed 0 and prefixed to the sentence. If we do not use a dummy root node, we instead return $\text{argmax}_i\, C[1][i][\leftarrow][1] + C[i][n][\rightarrow][1]$.

```
1  for $i : 0..n$ and all $d, c$
2      $C[i][i][d][c] \leftarrow 0.0$
3  for $m : 1..n$
4      for $i : 0..n-m$
5          $j \leftarrow i+m$
6          $C[i][j][\leftarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(j, i)$
7          $C[i][j][\rightarrow][0] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][1] + C[k+1][j][\leftarrow][1] + \textsc{Score}(i, j)$
8          $C[i][j][\leftarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\leftarrow][1] + C[k][j][\leftarrow][0]$
9          $C[i][j][\rightarrow][1] \leftarrow \max_{i \leq k < j} C[i][k][\rightarrow][0] + C[k][j][\rightarrow][1]$
10 return $C[0][n][\rightarrow][1]$
```

FIGURE 5. Eisner's cubic-time algorithm for arc-factored dependency parsing. Items of the form $C[i][j][d][c]$ represent subgraphs spanning from word i to j; $d = \leftarrow$ if the head is at the right periphery and $d = \rightarrow$ if the head is at the left periphery (the arrow pointing towards the dependents); $c = 1$ if the item is complete (that is, contains a head and its complete half-tree on the left/right) and $c = 0$ if the item is incomplete (that is, contains a head linked to a dependent with both inside half-trees).

what do we do if we want to allow non-projective trees as well? Finding a maximum spanning tree without the projectivity constraint is a harder problem that cannot be efficiently solved using dynamic programming. Fortunately, we can instead use standard algorithms from graph theory for finding a maximum spanning tree in a directed graph, such as the Chu-Liu-Edmonds algorithm (Chu & Liu, 1965; Edmonds, 1967).

This is a greedy recursive algorithm that starts by building a graph where every node (except the designated root node) has its highest-scoring incoming arc. If this graph is a tree, it must be the maximum spanning tree. If it is not a tree, it must contain at least one cycle. The algorithm then identifies a cycle, contracts the cycle into a single node and calls itself recursively on the smaller graph. The key idea is that a maximum spanning tree for the smaller contracted graph can be expanded to a maximum spanning tree for the larger initial graph, which means that when the recursion bottoms out (which it must do because the graphs keep getting smaller), we can construct successively larger solutions until we have the maximum spanning tree for the original graph. Every recursive call takes $O(n^2)$ time, and there can be at most $O(n)$ recursive calls, which means that the Chu-Liu-Edmonds algorithm can be used to perform non-projective dependency parsing with the arc-factored model in $O(n^3)$ time.[4] For the details of this algorithm, see McDonald et al. (2005b).

The use of spanning tree algorithms for non-projective spanning tree parsing was pioneered by McDonald et al. (2005b). Using the Chu-Liu-Edmonds algorithm together with an arc-factored model, they showed that parsing accuracy could be improved significantly for a language like Czech, where non-projective dependency trees are common, compared to a parser using the same model with Eisner's projective algorithm.

REFERENCES

Y. J. Chu & T. H. Liu (1965). 'On the Shortest Arborescence of a Directed Graph'. *Science Sinica* **14**:1396–1400.

[4]There is even an optimisation due to Tarjan (1977) that brings the complexity down to $O(n^2)$, paradoxically making non-projective parsing asymptotically faster than projective parsing with the arc-factored model.

M. Collins (2002). 'Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms'. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1–8.

J. Edmonds (1967). 'Optimum Branchings'. *Journal of Research of the National Bureau of Standards* **71B**:233–240.

J. M. Eisner (1996). 'Three new probabilistic models for dependency parsing: An exploration'. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pp. 340–345.

J. M. Eisner (2000). 'Bilexical grammars and their cubic-time parsing algorithms'. In H. Bunt & A. Nijholt (eds.), *Advances in Probabilistic and Other Parsing Technologies*, pp. 29–62. Kluwer.

R. McDonald (2006). *Discriminative Learning and Spanning Tree Algorithms for Dependency Parsing*. Ph.D. thesis, University of Pennsylvania.

R. McDonald, et al. (2005a). 'Online Large-Margin Training of Dependency Parsers'. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 91–98.

R. McDonald, et al. (2005b). 'Non-Projective Dependency Parsing using Spanning Tree Algorithms'. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pp. 523–530.

F. Rosenblatt (1958). 'The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain'. *Psychological Review* **65**(6):386–408.

R. E. Tarjan (1977). 'Finding Optimum Branchings'. *Networks* **7**:25–35.