Graph-based models for dependency parsing use a factorization defined in terms of the structure of the dependency graphs itself. In this lecture, I will introduce higher-order graph-based models (§1) and discuss different techniques for projective (§2) and non-projective (§3) parsing with these models.

1. Graph-Based Models

In the previous lecture, we looked at the arc-factored model for scoring a dependency tree $y$ for a given input $x$:

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

We also noted that this model is often used with unlabeled dependency trees, leaving the selection of labels for arcs as a later problem, which simplifies the model to:

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

We will stick with the unlabeled parsing problem throughout this lecture, as we will look at more complex factorisations of the dependency tree. For reasons of perspicuity, we will also suppress the sentence argument $x$ and use the more mnemonic symbols $h$ (for head) and $d$ (for dependent) instead of $i$ and $j$. Thus:

$$\text{Score}(x, y) = \sum_{(h, d) \in A_y} \text{Score}(h, d)$$

The arc-factored model is the simplest graph-based model for dependency parsing, a model that factorizes the scoring function in terms of subgraphs of the dependency tree. The arc-factored model is often referred to as a first-order model, because it decomposes the score of a tree into the scores of one arc at a time. From a linguistic point of view, this is a rather drastic assumption, similar to but even more severe than the independence assumptions of a standard PCFG. Not surprisingly, research has therefore shown that higher parsing accuracy can be achieved by taking

---

1 Date: 2013-03-08.

1 All the models discussed in this lecture exist in labeled versions, but they are rarely used in practice because they make learning and decoding more inefficient.
larger subgraphs of the dependency tree into account, leading to second- and third-order models where scores are defined over pairs or triples of arcs. However, when moving to higher-order models, it is important to manage the complexity so that we can still perform learning and decoding efficiently. In this section, we will discuss the models in themselves; in the next two sections, we will discuss how the models can be used for efficient processing of projective and non-projective dependency trees, respectively.

The first model we consider is the second-order sibling model originally due to Eisner (1996) but first used to achieve state-of-the-art accuracy with a global discriminative model by McDon-ald & Pereira (2006). In this model, we decompose trees into two-arc subgraphs consisting of a

---

**Table 1.** Graph-based model factors: first-order arc factors (top); second-order sibling and grand-child factors (middle); third-order tri-sibling and grand-sibling factors (bottom).
head word and two adjacent dependents on the same side of the head word. This type of structure is called a sibling factor, for obvious reasons, and is depicted schematically in the middle row of Table 1 in both its left-headed and right-headed version. When scoring dependency trees in this model, it is customary to include not only the more complex second-order sibling factors but also the simpler first-order arc factors from the arc-factored model (Table 1, top row), so that the relation between a head and dependent is scored both in isolation and in the context of the nearest preceding sibling $s$ (going inside out from the head):

$$ \text{Score}(x, y) = \sum_{(h, d) \in A_y} \text{Score}_1(h, d) + \sum_{(h, s, d) \in A_y} \text{Score}_2(h, s, d) $$

As usual, we assume that both of the simpler scoring functions take the form of a linear model:

$$ \text{Score}(x, y) = \sum_{(h, d) \in A_y} \sum_{k=1}^{K_1} f_k(h, d) \cdot w_k + \sum_{(h, s, d) \in A_y} \sum_{k=1}^{K_2} f_k(h, s, d) \cdot w_k $$

To make this a little bit more concrete, consider the dependency tree in Figure 1. The score of this tree under the second-order sibling model would be computed as follows (substituting word forms for tree nodes):

$$ \text{Score}(x, y) = \text{Score}_1(\text{root, sent}) + \text{Score}_1(\text{sent, She}) + \text{Score}_1(\text{sent, him}) + \text{Score}_1(\text{sent, message}) + \text{Score}_1(\text{sent, by}) + \text{Score}_1(\text{sent, .}) + \text{Score}_1(\text{message, a}) + \text{Score}_1(\text{by, email}) + \text{Score}_2(\text{root, , sent}) + \text{Score}_2(\text{sent, , She}) + \text{Score}_2(\text{sent, , him}) + \text{Score}_2(\text{sent, him, message}) + \text{Score}_2(\text{sent, message, by}) + \text{Score}_2(\text{sent, by, .}) + \text{Score}_2(\text{message, , by}) + \text{Score}_2(\text{by, , email}) $$

Sibling factors enable more complex feature templates, as exemplified in Table 2. Added on top of the first-order features considered in the previous lectures, they typically give a significant improvement in parsing accuracy because they capture relations between (adjacent) dependents of the same head. Another type of second-order factor, due to Carreras (2007), instead extends in the vertical dimension by including a dependent, its head, and the head of the head. These so-called grand-child factors are illustrated in the middle row of Table 1 (to the right). More recently, Koo & Collins (2010) have proposed the use of third-order graph-based models, incorporating both tri-sibling and grand-sibling factors (Table 1, bottom row). In their most complex model, which gives state-of-the-art accuracy for projective dependency parsing, the score of a dependency tree $y$ for a sentence $x$ is computed in the following way (where component scores come from a...
Table 2. Feature templates \( f_k(h, s, d) \) for the sibling factor \((h, s, d)\) in the context of sentence \( x \) (McDonald, 2006). Notation: \( w \)-word = word form of \( w \); \( w \)-pos = part-of-speech tag of \( w \). Binarization is used to obtain one binary feature for each possible instantiation of a template.

\[
\text{Score}(x, y) = \sum_{(h, d) \in A_y} \text{Score}_1(h, d) + \sum_{(h, s, d) \in A_y} \text{Score}_2s(h, s, d) + \sum_{(g, h, d) \in A_y} \text{Score}_2g(g, h, d) + \sum_{(g, h, s, d) \in A_y} \text{Score}_3g(g, h, s, d)
\]

For a discussion of feature templates used in this third-order model, see Koo & Collins (2010). Feature weights are typically learned using an online algorithm like the averaged perceptron, as described in the previous lecture.

2. Projective Parsing

If we restrict our search space to projective trees, then decoding for higher-order models can be performed with adaptations of Eisner’s algorithm that we discussed in the last lecture. In fact, the choice of which higher-order factors to include in the models is to a large extent influenced by the desire to perform efficient parsing with this type of algorithm. For example, all the factors illustrated in Table 1 have in common that they only consider dependents on one side of the head in order to be compatible with the split-head representation used in Eisner’s algorithm. Below we describe the adaptation of Eisner’s algorithm for the second-order sibling model of McDonald & Pereira (2006), which still runs in \( O(n^3) \) time. For the more complex models, which require \( O(n^4) \) time because of an extra loop (either over the grandparent or the second sibling), we refer to Koo & Collins (2010).

The second-order sibling version of Eisner’s algorithm is specified in pseudo-code in Figure 2. The initialization (lines 1–2) and the construction of complete items (lines 11–12) are exactly as before, but incomplete items need to be constructed differently (lines 6–10). First of all, we introduce a new type of item called a sibling item with the signature \( C[i][j][−][2] \). A sibling item is similar to an incomplete item in that it consists of two adjacent half-trees (complete items), the first with its head to the left and the second with its head to the right. The difference is that there is no arc connecting the two heads because they are both assumed to be dependents of a third word (as yet unspecified). A sibling item from \( i \) to \( j \) is formed from two complete items by finding the split point \( k \) that maximizes the sum of the scores of the two items combined (line 6). Incomplete items can then be formed in one of two ways. First, we can combine a head with no previous dependents with a complete item on its left (line 7) or on its right (line 8), in which case the resulting score is the sum of the scores of the two items plus the score of the sibling factor with no preceding sibling. Secondly, we can combine an incomplete item with a sibling item on its left (line 9) or on its right (line 10) by finding the split point \( k \) that maximizes the
for \( i : 0..n \) and all \( d, c \)
\[
\begin{align*}
\text{C}[i][j][d][c] &\leftarrow 0.0 \\
\text{for } m : 1..n \\
\text{for } i : 0..n-m \\
\text{\( j \rightarrow i + m \)} \\
\text{\( C[i][j][-][2] \leftarrow \max_{k<j} C[i][k][-][1] + C[k+1][j][-][1] \)} \\
\text{\( C[i][j][-][0] \leftarrow C[i][j-1][-][1] + C[j][j][-][1] + \text{Score}(j, -i) \)} \\
\text{\( C[i][j][0][0] \leftarrow C[i][i][0][1] + C[i+1][j][-][1] + \text{Score}(i, -j) \)} \\
\text{\( C[i][j][-][0] \leftarrow \max\{C[i][j][-][0], \max_{k<j} C[i][k][-][2] + C[k][j][-][0] + \text{Score}(j, k, i)\} \)} \\
\text{\( C[i][j][0][0] \leftarrow \max\{C[i][j][0][0], \max_{k<j} C[i][k][-][1] + C[k][j][-][0] + \text{Score}(i, k, j)\} \)} \\
\text{\( C[i][j][-][1] \leftarrow \max_{k<j} C[i][k][-][1] + C[k][j][-][0] \)} \\
\text{\( C[i][j][0][1] \leftarrow \max_{k<j} C[i][k][0][1] + C[k][j][-][1] \)} \\
\text{return } C[0][n][-][1]
\end{align*}
\]

Figure 2. Eisner’s cubic-time algorithm for dependency parsing with a second-order sibling model. 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 left periphery, \( d = \rightarrow \) if the head is at the right periphery (the arrow pointing towards the dependents), and \( d = \) if there is no head (sibling item); \( c = 0 \) if the item is an incomplete head item (that is, contains a head linked to a dependent with both inside half-trees), \( c = 1 \) if the item is a complete head item (that is, contains a head and its complete half-tree on the left/right), and \( c = 2 \) if the item is a sibling item (that is, contains two adjacent siblings with both inside half-trees).

sum of the scores of the items plus the score of the sibling factor where \( k \) is the first sibling as well as the dependent of the incomplete item. However, we must also check that this score is better than the previous best incomplete item for this span, which could be an incomplete item with a single dependent (formed in line 7 or 8). Finally, we return the highest scoring complete item spanning from 0 to \( n \) with its head to the left as before. The main part of the algorithm has two nested for loops plus a number of (non-nested) maximization loops, which means that parsing complexity remains \( O(n^3) \).

3. Non-Projective Parsing

In the previous lecture, we saw how graph-theoretic algorithms like Chu-Liu-Edmonds can be used for efficient non-projective dependency parsing. Unfortunately, these methods do not generalize to higher-order models. On the contrary, it has been shown that non-projective parsing with higher-order models is NP hard (McDonald & Satta, 2007), which means that we are not likely to find any algorithms that run in polynomial time (that is, \( O(n^k) \) time for some constant \( k \), as opposed to exponential time, which is \( O(k^n) \)).

Since higher-order models clearly improve parsing accuracy, recent research has focused on finding good approximate decoding algorithms, that is, algorithms that are not guaranteed to return the highest scoring dependency tree relative to the model but are likely to return a good (or even optimal) tree in a high proportion of cases. The first work in this line of research is McDonald & Pereira (2006), who proposes a two-step approach where the first step is to find the best projective tree (using Eisner’s algorithm) and the second step is to iteratively substitute arcs in the tree as long as the score of the tree improves. Since the first step is guaranteed to return the highest-scoring projective tree, any tree that has a higher score must contain at
At least one non-projective arc. Although the second step could in the worst case take exponential time (enumerating all possible non-projective trees), it usually converges very quickly in practice. This approximate second-order parsing model for non-projective trees usually gives higher parsing accuracy than the exact first-order model using spanning tree parsing. However, it is crucial that the weights for the scoring model are learned online with the full two-step decoder. The reason is that the best projective tree returned in the first step should not necessarily be the one that would be returned by a model trained only on projective trees, but rather a tree that is likely to lead to a high-scoring non-projective tree after arc-substitution.

Other approaches to approximate higher-order non-projective dependency parsing are based on general optimization techniques such as integer linear programming (ILP) (Riedel & Clarke, 2006; Martins et al., 2009), belief propagation (Smith & Eisner, 2008), and dual decomposition (Koo et al., 2010). Dual decomposition is a technique for optimization of hard problems through joint optimization of simpler problems for which exact algorithms exist. For example, in Koo et al. (2010), the problem of higher-order non-projective dependency parsing is modeled as the joint optimization of two model scores:

\[ y^*, z^* = \arg\max_{y \in \text{GEN}(x), z \in \text{DG}(x), y \neq z} \text{Score}_1(x, y) + \text{Score}_2(x, z) \]

The first model is an arc-factored model, where the highest-scoring dependency tree \( y \in \text{GEN}(x) \) can be computed efficiently using the Chu-Liu-Edmonds algorithm (and where any analysis returned is guaranteed to be a tree). The second model is a higher-order model (using second- and third-order factors as discussed in §1), where the highest scoring tree cannot be computed efficiently but where we can use simple head automata to find the highest scoring set of dependents for each word, which gives us a dependency graph \( z \in \text{DG}(x) \) is the set of all dependency graphs for \( x \) without imposing the tree constraint). The key idea is now that by constraining \( y \) and \( z \) to be the same, we get an analysis that has to be a tree (thanks to the tree constraint imposed by \( \text{Score}_1(x, y) \)) and is likely to be a good analysis (thanks to the higher-order factors inherent in \( \text{Score}_2(x, z) \)). The dual decomposition algorithm tries to find such a solution by first computing \( y^* = \arg\max_{y \in \text{GEN}(x)} \text{Score}_1(x, y) \) and \( z^* = \arg\max_{z \in \text{DG}(x)} \text{Score}_2(x, y) \) separately, using the two exact decoders. If \( y^* = z^* \), then we have found the optimal dependency tree. If not, we impose penalty weights on arcs that are in \( y^* \) but not in \( z^* \) and vice versa, so that the two solutions computed separately are more likely to be the same next time. We then iterate these steps until we either find a solution or reach some pre-specified limit. The dual decomposition algorithm is approximate in the sense that it is not guaranteed to find the exact solution, but it differs from many other approximate algorithms in that it is always possible to tell whether the solution found is exact or not. And in experiments reported by Koo et al. (2010), the exact solution was found for over 95% of the sentences in test sets for a wide range of languages.

References


\(^2\)An interesting variant on this technique is to add arcs instead of substituting them, which gives a parser for directed acyclic graphs instead of trees; see McDonald & Pereira (2006) for more details.

\(^3\)ILP is in principle an exact method, but decoding usually has to rely on a relaxation to linear programming (LP) that may not return exact solutions to the original ILP problem. Similarly, belief propagation can be exact but is not for the belief networks used for non-projective dependency parsing because of cycles in the factor graph.


