Dependency Parsing 2
CMSC 723 / LING 723 / INST 725

Marine Carpuat

Fig credits: Joakim Nivre, Dan Jurafsky & James Martin
Dependency Parsing

• Formalizing dependency trees

• Transition-based dependency parsing
  • Shift-reduce parsing
  • Transition system
  • Oracle
  • Learning/predicting parsing actions
Data-driven dependency parsing

**Goal:** learn a good predictor of dependency graphs
Input: sentence
Output: dependency graph/tree $G = (V,A)$

Can be framed as a structured prediction task
- very large output space
- with interdependent labels

2 dominant approaches: transition-based parsing and graph-based parsing
Transition-based dependency parsing

- Builds on shift-reduce parsing [Aho & Ullman, 1927]

- **Configuration**
  - Stack
  - Input buffer of words
  - Set of dependency relations

- **Goal of parsing**
  - find a final configuration where
  - all words accounted for
  - Relations form dependency tree
Transition operators

• Transitions: produce a new configuration given current configuration

• Parsing is the task of
  • Finding a sequence of transitions
  • That leads from start state to desired goal state

• Start state
  • Stack initialized with ROOT node
  • Input buffer initialized with words in sentence
  • Dependency relation set = empty

• End state
  • Stack and word lists are empty
  • Set of dependency relations = final parse
Arc Standard Transition System

• Defines 3 transition operators [Covington, 2001; Nivre 2003]

• LEFT-ARC:
  • create head-dependent rel. between word at top of stack and 2\textsuperscript{nd} word (under top)
  • remove 2\textsuperscript{nd} word from stack

• RIGHT-ARC:
  • Create head-dependent rel. between word on 2\textsuperscript{nd} word on stack and word on top
  • Remove word at top of stack

• SHIFT
  • Remove word at head of input buffer
  • Push it on the stack
Arc standard transition systems

• Preconditions
  • ROOT cannot have incoming arcs
  • LEFT-ARC cannot be applied when ROOT is the 2\textsuperscript{nd} element in stack
  • LEFT-ARC and RIGHT-ARC require 2 elements in stack to be applied
Transition-based Dependency Parser

- Assume an oracle
- Parsing complexity
  - Linear in sentence length!
- Greedy algorithm
  - Unlike Viterbi for POS tagging

```plaintext
function DEPENDENCYPARSE(words) returns dependency tree
    state ← {{root}, [words], []} ; initial configuration
    while state not final
        t ← ORACLE(state) ; choose a transition operator to apply
        state ← APPLY(t, state) ; apply it, creating a new state
    return state
```

Figure 14.6 A generic transition-based dependency parser
Transition-Based Parsing Illustrated

```
  root
  
  obj
  
  det
  
  nmod

Book me the morning flight
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word List</th>
<th>Action</th>
<th>Relation Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[root]</td>
<td>[book, me, the, morning, flight]</td>
<td>SHIFT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[root, book]</td>
<td>[me, the, morning, flight]</td>
<td>SHIFT</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[root, book, me]</td>
<td>[the, morning, flight]</td>
<td>RIGHTARC</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[root, book]</td>
<td>[the, morning, flight]</td>
<td>SHIFT</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[root, book, the]</td>
<td>[morning, flight]</td>
<td>SHIFT</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[root, book, the, morning]</td>
<td>[flight]</td>
<td>SHIFT</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[root, book, the, morning, flight]</td>
<td>[]</td>
<td>LEFTARC</td>
<td>(book → me)</td>
</tr>
<tr>
<td>7</td>
<td>[root, book, the, flight]</td>
<td>[]</td>
<td>LEFTARC</td>
<td>(morning ← flight)</td>
</tr>
<tr>
<td>8</td>
<td>[root, book, flight]</td>
<td>[]</td>
<td>RIGHTARC</td>
<td>(book → flight)</td>
</tr>
<tr>
<td>10</td>
<td>[root]</td>
<td>[]</td>
<td>Done</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 14.7** Trace of a transition-based parse.
Where to we get an oracle?

• Multiclass classification problem
  • Input: current parsing state (e.g., current and previous configurations)
  • Output: one transition among all possible transitions
  • Q: size of output space?

• Supervised classifiers can be used
  • E.g., perceptron

• Open questions
  • What are good features for this task?
  • Where do we get training examples?
Generating Training Examples

- What we have in a treebank
- What we need to train an oracle
  - Pairs of configurations and predicted parsing action

![Diagram](image)

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word List</th>
<th>Predicted Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[root]</td>
<td>[book, the, flight, through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>1</td>
<td>[root, book]</td>
<td>[the, flight, through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>2</td>
<td>[root, book, the]</td>
<td>[flight, through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>3</td>
<td>[root, book, the, flight]</td>
<td>[through, houston]</td>
<td>LEFTARC</td>
</tr>
<tr>
<td>4</td>
<td>[root, book, flight]</td>
<td>[through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>5</td>
<td>[root, book, flight, through]</td>
<td>[houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>6</td>
<td>[root, book, flight, through, houston]</td>
<td>[]</td>
<td>LEFTARC</td>
</tr>
<tr>
<td>7</td>
<td>[root, book, flight, houston ]</td>
<td>[]</td>
<td>RIGHTARC</td>
</tr>
<tr>
<td>8</td>
<td>[root, book, flight]</td>
<td>[]</td>
<td>RIGHTARC</td>
</tr>
<tr>
<td>9</td>
<td>[root, book]</td>
<td>[]</td>
<td>RIGHTARC</td>
</tr>
<tr>
<td>10</td>
<td>[root]</td>
<td>[]</td>
<td>Done</td>
</tr>
</tbody>
</table>

**Figure 14.8** Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.
Generating training examples

• Approach: simulate parsing to generate reference tree

• Given
  • A current config with stack S, dependency relations Rc
  • A reference parse (V,Rp)

• Do

\[
\text{LEFTARC}(r): \text{if } (S_1 \ r \ S_2) \in R_p \\
\text{RIGHTARC}(r): \text{if } (S_2 \ r \ S_1) \in R_p \text{ and } \forall r', w \text{ s.t. } (S_1 \ r' \ w) \in R_p \text{ then } (S_1 \ r' \ w) \in R_c \\
\text{SHIFT}: \text{otherwise}
\]
Let’s try it out

**LEFTARC(r):** If \((S_1 \ r \ S_2) \in R_p\)

**RIGHTARC(r):** If \((S_2 \ r \ S_1) \in R_p\) and \(\forall r', w\) s.t. \((S_1 \ r' \ w) \in R_p\) then \((S_1 \ r' \ w) \in R_c\)

**SHIFT:** otherwise

```
root
  /   |
  dobj
 /   |
  det

/   |
nmod

/   |
case

Book the flight through Houston
```
Features

• Configuration consist of stack, buffer, current set of relations

• Typical features
  • Features focus on top level of stack
  • Use word forms, POS, and their location in stack and buffer
Features example

• Given configuration

<table>
<thead>
<tr>
<th>Stack</th>
<th>Word buffer</th>
<th>Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root, canceled, flights]</td>
<td>[to Houston]</td>
<td>(canceled $\rightarrow$ United)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(flights $\rightarrow$ morning)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(flights $\rightarrow$ the)</td>
</tr>
</tbody>
</table>

• Example of useful features

\begin{align*}
\langle s_1.w = \text{flights}, op = \text{shift} \rangle \\
\langle s_2.w = \text{canceled}, op = \text{shift} \rangle \\
\langle s_1.t = \text{NNS}, op = \text{shift} \rangle \\
\langle s_2.t = \text{VBD}, op = \text{shift} \rangle \\
\langle b_1.w = \text{to}, op = \text{shift} \rangle \\
\langle b_1.t = \text{TO}, op = \text{shift} \rangle \\
\langle s_1.wt = \text{flightsNNS}, op = \text{shift} \rangle \\
\langle s_1.t \cdot s_2.t = \text{NNSVBD}, op = \text{shift} \rangle
\end{align*}
### Features example

<table>
<thead>
<tr>
<th>Source</th>
<th>Feature templates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One word</strong></td>
<td>( s_1.w )</td>
</tr>
<tr>
<td>( s_2.w )</td>
<td>( s_2.t )</td>
</tr>
<tr>
<td>( b_1.w )</td>
<td>( b_1.w )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>Feature templates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two word</strong></td>
<td>( s_1.w \circ s_2.w )</td>
</tr>
<tr>
<td>( s_1.t \circ s_2.wt )</td>
<td>( s_1.w \circ s_2.w \circ s_2.t )</td>
</tr>
<tr>
<td>( s_1.w \circ s_1.t \circ s_2.t )</td>
<td>( s_1.w \circ s_1.t )</td>
</tr>
</tbody>
</table>

**Figure 14.9** Standard feature templates for training transition-based dependency parsers. In the template specifications \( s_n \) refers to a location on the stack, \( b_n \) refers to a location in the word buffer, \( w \) refers to the wordform of the input, and \( t \) refers to the part of speech of the input.
Research highlight: Dependency parsing with stack-LSTMs

• From Dyer et al. 2015: http://www.aclweb.org/anthology/P15-1033

• Idea
  • Instead of hand-crafted feature
  • Predict next transition using recurrent neural networks to learn representation of stack, buffer, sequence of transitions
Research highlight: Dependency parsing with stack-LSTMs

Figure 1: A stack LSTM extends a conventional left-to-right LSTM with the addition of a stack pointer (notated as TOP in the figure). This figure shows three configurations: a stack with a single element (left), the result of a `pop` operation to this (middle), and then the result of applying a `push` operation (right). The boxes in the lowest rows represent stack contents, which are the inputs to the LSTM, the upper rows are the outputs of the LSTM (in this paper, only the output pointed to by TOP is ever accessed), and the middle rows are the memory cells (the c_t’s and h_t’s) and gates. Arrows represent function applications (usually affine transformations followed by a nonlinearity), refer to §2.1 for specifics.
Research highlight:
Dependency parsing with stack-LSTMs

Figure 2: Parser state computation encountered while parsing the sentence “an overhasty decision was made.” Here $S$ designates the stack of partially constructed dependency subtrees and its LSTM encoding; $B$ is the buffer of words remaining to be processed and its LSTM encoding; and $A$ is the stack representing the history of actions taken by the parser. These are linearly transformed, passed through a ReLU nonlinearity to produce the parser state embedding $p_t$. An affine transformation of this embedding is passed to a softmax layer to give a distribution over parsing decisions that can be taken.
Alternate Transition Systems
Note: A different way of writing arc-standard transition system

- **Transitions:**
  - **Left-Arc\(_k\):**
    \[ (\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \cup \{(j, i, k)\}) \]
  - **Right-Arc\(_k\):**
    \[ (\sigma|i, j|\beta, A) \Rightarrow (\sigma, i|\beta, A \cup \{(i, j, k)\}) \]
  - **Shift:**
    \[ (\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A) \]

- **Preconditions:**
  - **Left-Arc\(_k\):**
    \[ \neg[i = 0] \]
    \[ \neg\exists i' \exists k'[(i', i, k') \in A] \]
  - **Right-Arc\(_k\):**
    \[ \neg\exists i' \exists k'[(i', j, k') \in A] \]
A weakness of arc-standard parsing

Right dependents cannot be attached to their head until all their dependents have been attached.

![Diagram](image.png)

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word List</th>
<th>Predicted Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[root]</td>
<td>[book, the, flight, through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>1</td>
<td>[root, book]</td>
<td>[the, flight, through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>2</td>
<td>[root, book, the]</td>
<td>[flight, through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>3</td>
<td>[root, book, the, flight]</td>
<td>[through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>4</td>
<td>[root, book, flight]</td>
<td>[through, houston]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>5</td>
<td>[root, book, flight, through]</td>
<td>[through, houston]</td>
<td>LEFTARC</td>
</tr>
<tr>
<td>6</td>
<td>[root, book, flight, through, houston]</td>
<td>[]</td>
<td>SHIFT</td>
</tr>
<tr>
<td>7</td>
<td>[root, book, flight, through, houston]</td>
<td>[]</td>
<td>RIGHTARC</td>
</tr>
<tr>
<td>8</td>
<td>[root, book, flight]</td>
<td>[]</td>
<td>RIGHTARC</td>
</tr>
<tr>
<td>9</td>
<td>[root, book]</td>
<td>[]</td>
<td>RIGHTARC</td>
</tr>
<tr>
<td>10</td>
<td>[root]</td>
<td>[]</td>
<td>Done</td>
</tr>
</tbody>
</table>

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.
Arc Eager Parsing

• LEFT-ARC:
  • Create head-dependent rel. between word at front of buffer and word at top of stack
  • pop the stack

• RIGHT-ARC:
  • Create head-dependent rel. between word on top of stack and word at front of buffer
  • Shift buffer head to stack

• SHIFT
  • Remove word at head of input buffer
  • Push it on the stack

• REDUCE
  • Pop the stack
## Arc Eager Parsing Example

![Diagram of book the flight through Houston](image)

### Table: Processing Trace

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word List</th>
<th>Action</th>
<th>Relation Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[root]</td>
<td>[book, the, flight, through, houston]</td>
<td>RIGHTARC</td>
<td>(root → book)</td>
</tr>
<tr>
<td>1</td>
<td>[root, book]</td>
<td>[the, flight, through, houston]</td>
<td>SHIFT</td>
<td>(the ← flight)</td>
</tr>
<tr>
<td>2</td>
<td>[root, book, the]</td>
<td>[flight, through, houston]</td>
<td>LEFTARC</td>
<td>(book → flight)</td>
</tr>
<tr>
<td>3</td>
<td>[root, book]</td>
<td>[flight, through, houston]</td>
<td>RIGHTARC</td>
<td>(through ← houston)</td>
</tr>
<tr>
<td>4</td>
<td>[root, book, flight]</td>
<td>[through, houston]</td>
<td>SHIFT</td>
<td>(flight → houston)</td>
</tr>
<tr>
<td>5</td>
<td>[root, book, flight, through]</td>
<td>[houston]</td>
<td>LEFTARC</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[root, book, flight]</td>
<td>[houston]</td>
<td>RIGHTARC</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[root, book, flight, houston]</td>
<td>[]</td>
<td>REDUCE</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[root, book, flight]</td>
<td>[]</td>
<td>REDUCE</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>[root, book]</td>
<td>[]</td>
<td>REDUCE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[root]</td>
<td>[]</td>
<td>REDUCE</td>
<td>Done</td>
</tr>
</tbody>
</table>

**Figure 14.10** A processing trace of *Book the flight through Houston* using the arc-eager transition operators.
Trees & Forests

- A dependency forest (here) is a dependency graph satisfying
  - Root
  - Single-Head
  - Acyclicity
  - but **not** Connectedness
Properties of this transition-based parsing algorithm

- Correctness
  - For every complete transition sequence, the resulting graph is a projective dependency forest (soundness)
  - For every projective dependency forest $G$, there is a transition sequence that generates $G$ (completeness)

- Trick: forest can be turned into tree by adding links to ROOT₀
Dealing with non-projectivity
Projectivity

• **Arc** from head to dependent is **projective**
  • If there is a path from head to every word between head and dependent

• **Dependency tree** is **projective**
  • If all arcs are projective
  • Or equivalently, if it can be drawn with no crossing edges

• Projective trees make computation easier
• But most theoretical frameworks do not assume projectivity
  • Need to capture long-distance dependencies, free word order
Arc-standard parsing can’t produce non-projective trees
Pred

Atr

[root Z]  nich [je jen jedna na kvalitu .]
(out-of) (them) (is) (only) (one) (to) (quality)
How frequent are non-projective structures?

- Statistics from CoNLL shared task
  - NPD = non projective dependencies
  - NPS = non projective sentences

<table>
<thead>
<tr>
<th>Language</th>
<th>%NPD</th>
<th>%NPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dutch</td>
<td>5.4</td>
<td>36.4</td>
</tr>
<tr>
<td>German</td>
<td>2.3</td>
<td>27.8</td>
</tr>
<tr>
<td>Czech</td>
<td>1.9</td>
<td>23.2</td>
</tr>
<tr>
<td>Slovene</td>
<td>1.9</td>
<td>22.2</td>
</tr>
<tr>
<td>Portuguese</td>
<td>1.3</td>
<td>18.9</td>
</tr>
<tr>
<td>Danish</td>
<td>1.0</td>
<td>15.6</td>
</tr>
</tbody>
</table>
How to deal with non-projectivity?

(1) change the transition system

<table>
<thead>
<tr>
<th>Transition</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP-Left$_r$</td>
<td>$(\sigma</td>
</tr>
<tr>
<td>NP-Right$_r$</td>
<td>$(\sigma</td>
</tr>
</tbody>
</table>

- Add new transitions
  - That apply to 2$^{nd}$ word of the stack
  - Top word of stack is treated as context

[Attardi 2006]
How to deal with non-projectivity?
(2) pseudo-projective parsing

Solution:
• “projectivize” a non-projective tree by creating new projective arcs
• That can be transformed back into non-projective arcs in a post-processing step
How to deal with non-projectivity? (2) pseudo-projective parsing

Solution:
• "projectivize" a non-projective tree by creating new projective arcs
• That can be transformed back into non-projective arcs in a post-processing step
Graph-based parsing
A graph $G = (V, A)$ is a set of vertices $V$ and arcs $(i,j) \in A$, where $i,j \in V$

Undirected graphs: $(i,j) \in A \iff (j,i) \in A$

Directed graphs (digraphs): $(i,j) \in A \not\Rightarrow (j,i) \in A$
Directed Spanning Trees

- A directed spanning tree of a (multi-)digraph \( G = (V, A) \), is a subgraph \( G' = (V', A') \) such that:
  - \( V' = V \)
  - \( A' \subseteq A \), and \( |A'| = |V'| - 1 \)
  - \( G' \) is a tree (acyclic)

- A spanning tree of the following (multi-)digraphs
Maximum Spanning Tree

• Assume we have an arc factored model
  i.e. weight of graph can be factored as sum or product of weights of its arcs

• Chu-Liu-Edmonds algorithm can find the maximum spanning tree for us!
  • Greedy recursive algorithm
  • Naïve implementation: O(n^3)
Chu-Liu-Edmonds illustrated
Chu-Liu-Edmonds illustrated

- Find highest scoring incoming arc for each vertex

- If this is a tree, then we have found MST!!
Chu-Liu-Edmonds illustrated

- If not a tree, identify cycle and contract
- Recalculate arc weights into and out-of cycle
Chu-Liu-Edmonds illustrated

- Outgoing arc weights
  - Equal to the max of outgoing arc over all vertexes in cycle
  - e.g., John → Mary is 3 and saw → Mary is 30
Chu-Liu-Edmonds illustrated

- **Incoming arc weights**
  - Equal to the weight of best spanning tree that includes head of incoming arc, and all nodes in cycle
  - root → saw → John is 40 (**)
  - root → John → saw is 29
This is a tree and the MST for the contracted graph!!

Go back up recursive call and reconstruct final graph
Arc weights as linear classifiers

\[ w_{ij}^k = e^{\mathbf{w} \cdot \mathbf{f}(i,j,k)} \]

- Arc weights are a linear combination of features of the arc, \( \mathbf{f} \), and a corresponding weight vector \( \mathbf{w} \)
- Raised to an exponent (simplifies some math ...)
- What arc features?
- [McDonald et al. 2005] discuss a number of binary features
Example of classifier features

Features from [McDonald et al. 2005]:

- Identities of the words $w_i$ and $w_j$ and the label $l_k$

head = saw & dependent = with
How to score a graph $G$ using features?

By definition of arc weights as linear classifiers

$$G = \arg \max_{G \in T(G_x)} \prod_{(i,j,k) \in G} w^k_{ij} = \arg \max_{G \in T(G_x)} \prod_{(i,j,k) \in G} e^{w \cdot f(i,j,k)}$$

$$= \arg \max_{G \in T(G_x)} \log \prod_{(i,j,k) \in G} e^{w \cdot f(i,j,k)}$$

$$= \arg \max_{G \in T(G_x)} \sum_{(i,j,k) \in G} w \cdot f(i,j,k)$$

$$= \arg \max_{G \in T(G_x)} w \cdot \sum_{(i,j,k) \in G} f(i,j,k) = \arg \max_{G \in T(G_x)} w \cdot f(G)$$
How can we learn the classifier from data?

e.g., The Perceptron

Training data: \( \mathcal{T} = \{(x_t, G_t)\}_{t=1}^{\mathcal{T}} \)

1. \( \mathbf{w}^{(0)} = 0; \ i = 0 \)
2. for \( n: 1..N \)
3. for \( t: 1..T \)
4. Let \( G' = \arg\max_{G'} \mathbf{w}^{(i)} \cdot f(G') \)
5. if \( G' \neq G_t \)
6. \( \mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + f(G_t) - f(G') \)
7. \( i = i + 1 \)
8. return \( \mathbf{w}' \)
Dependency Parsing: what you should know

• Formalizing dependency trees

• Transition-based dependency parsing
  • Shift-reduce parsing
  • Transition system: arc standard, arc eager
  • Oracle
  • Learning/predicting parsing actions

• Graph-based dependency parsing

• A flexible framework that allows many extensions
  • RNNs vs feature engineering, non-projectivity
Extension: dynamic oracle

Problem with standard classifier-based oracle:
- It is “static”
  - ie tied to optimal config sequence that produces gold tree
- What if there are multiple sequences for a single gold tree?
- How can we recover if the parser deviates from gold sequence?

One solution: “dynamic oracle” [Goldberg & Nivre 2012]

See also Locally Optimal Learning to Search [Chang et al. ICML 2015]
Extension: dynamic oracle

Algorithm 3 Online training with a dynamic oracle

1: \( w \leftarrow 0 \)
2: for \( I = 1 \to \text{ITERATIONS} \) do
3:   for sentence \( x \) with gold tree \( G_{\text{gold}} \) in corpus do
4:     \( c \leftarrow c_s(x) \)
5:     while \( c \) is not terminal do
6:       \( t_p \leftarrow \arg \max_t w \cdot \phi(c, t) \)
7:       \[
        \text{ZERO\_COST} \leftarrow \{t | o(t; c, G_{\text{gold}}) = \text{true}\}
      \]
8:       \( t_o \leftarrow \arg \max_{t \in \text{ZERO\_COST}} w \cdot \phi(c, t) \)
9:       if \( t_p \not\in \text{ZERO\_COST} \) then
10:      \( w \leftarrow w + \phi(c, t_o) - \phi(c, t_p) \)
11: t_n \leftarrow \text{CHOOSE\_NEXT}(I, t_p, \text{ZERO\_COST})
12: c \leftarrow t_n(c)
13: return \( w \)

See [Goldberg & Nivre 2012] for details