# Training Deterministic Parsers with Non-Deterministic Oracles

**Yoav Goldberg**
Bar-Ilan University
Department of Computer Science
Ramat-Gan, Israel
yoav.goldberg@gmail.com

**Joakim Nivre**
Uppsala University
Department of Linguistics and Philology
Uppsala, Sweden
joakim.nivre@lingfil.uu.se

## Abstract

Greedy transition-based parsers are very fast but tend to suffer from error propagation. This problem is aggravated by the fact that they are normally trained using oracles that are deterministic and incomplete in the sense that they assume a unique canonical path through the transition system and are only valid as long as the parser does not stray from this path. In this paper, we give a general characterization of oracles that are nondeterministic and complete, present a method for deriving such oracles for transition systems that satisfy a property we call arc decomposition, and instantiate this method for three well-known transition systems from the literature. We say that these oracles are dynamic, because they allow us to dynamically explore alternative and non-optimal paths during training – in contrast to oracles that statically assume a unique optimal path. Experimental evaluation on a wide range of data sets clearly shows that using dynamic oracles to train greedy parsers gives substantial improvements in accuracy. Moreover, this improvement comes at no cost in terms of efficiency, unlike other techniques like beam search.

## 1 Introduction

Greedy transition-based parsers are easy to implement and are very efficient, but they are generally not as accurate as parsers that are based on global search (McDonald et al., 2005; Koo and Collins, 2010) or as transition-based parsers that use beam search (Zhang and Clark, 2008) or dynamic programming (Huang and Sagae, 2010; Kuhlmann et al., 2011). This work is part of a line of research trying to push the boundaries of greedy parsing and narrow the accuracy gap of 2–3% between search-based and greedy parsers, while maintaining the efficiency and incremental nature of greedy parsers.

One reason for the lower accuracy of greedy parsers is error propagation: once the parser makes an error in decoding, more errors are likely to follow. This behavior is closely related to the way in which greedy parsers are normally trained. Given a treebank oracle, a gold sequence of transitions is derived, and a predictor is trained to predict transitions along this gold sequence, without considering any parser state outside this sequence. Thus, once the parser strays from the golden path at test time, it ventures into unknown territory and is forced to react to situations it has never been trained for.

In recent work (Goldberg and Nivre, 2012), we introduced the concept of a *dynamic* oracle, which is non-deterministic and not restricted to a single golden path, but instead provides optimal predictions for any possible state the parser might be in. Dynamic oracles are non-deterministic in the sense that they return a *set* of valid transitions for a given parser state and gold tree. Moreover, they are well-defined and optimal also for states from which the gold tree cannot be derived, in the sense that they return the set of transitions leading to the best tree derivable from each state. We showed experimentally that, using a dynamic oracle for the arc-eager transition system (Nivre, 2003), a greedy parser can be trained to perform well also after incurring a mistake, thus alleviating the effect of error propagation and resulting in consistently better parsing accuracy.

Transactions of the Association for Computational Linguistics, 1 (2013) 403–414. Action Editor: Jason Eisner.
Submitted 6/2013; Published 10/2013.

In this paper, we extend the work of Goldberg and Nivre (2012) by giving a general characterization of dynamic oracles as oracles that are *non-deterministic*, in that they return sets of transitions, and *complete*, in that they are defined for all possible states. We then define a formal property of transition systems which we call *arc decomposition*, and introduce a framework for deriving dynamic oracles for arc-decomposable systems. Using this framework, we derive novel dynamic oracles for the hybrid (Kuhlmann et al., 2011) and easy-first (Goldberg and Elhadad, 2010) transition systems, which are arc-decomposable (as is the arc-eager system). We also show that the popular arc-standard system (Nivre, 2004) is not arc-decomposable, and so deriving a dynamic oracle for it remains an open research question. Finally, we perform a set of experiments on the CoNLL 2007 data sets, validating that the use of dynamic oracles for exploring states that result from parsing mistakes during training is beneficial across transition systems.

## 2 Transition-Based Dependency Parsing

We begin with a quick review of transition-based dependency parsing, presenting the arc-eager, arc-standard, hybrid and easy-first transitions systems in a common notation. The transition-based parsing framework (Nivre, 2008) assumes a *transition system*, an abstract machine that processes sentences and produces parse trees. The transition system has a set of *configurations* and a set of *transitions* which are applied to configurations. When parsing a sentence, the system is initialized to an *initial configuration* based on the input sentence, and transitions are repeatedly applied to this configuration. After a finite number of transitions, the system arrives at a *terminal configuration*, and a parse tree is read off the terminal configuration. In a greedy parser, a classifier is used to choose the transition to take in each configuration, based on features extracted from the configuration itself. Transition systems differ by the way they define configurations, and by the particular set of transitions available.

### 2.1 Dependency Trees

We define a *dependency tree* for a sentence $W = w_1, \ldots, w_n$ to be a labeled directed tree $T = (V, A)$, where $V = \{w_1, \ldots, w_n\}$ is a set of nodes given by the tokens of the input sentence, and $A \subseteq V \times L \times V$ (for some dependency label set $L$) is a set of labeled directed arcs of the form $(h, lb, d)$, where $h \in V$ is said to be the head, $d \in V$ the dependent, and $lb \in L$ the dependency label.

When dealing with unlabeled parsing, or when the label identity is irrelevant, we take $A \subseteq V \times V$ to be a set of ordinary directed arcs of the form $(h, d)$. Note that, since the nodes of the tree are given by the input sentence, a dependency tree $T = (V, A)$ for a sentence $W$ is uniquely defined by the arc set $A$. For convenience, we will therefore equate the tree with the arc set and and use the symbol $T$ for the latter, reserving the symbol $A$ for arc sets that are not necessarily trees. In the context of this work it is assumed that all the dependency trees are *projective*.

Although the general definition of a dependency tree does not make any assumptions about which node is the root of the tree, it is common practice in dependency parsing to add a dummy node ROOT, which is prefixed or suffixed to the sentence and which always acts as the root of the tree. We will follow this practice in our description of different transition systems below.

### 2.2 Transition Systems

**Arc-Eager** In the arc-eager system (Nivre, 2003), a configuration $c = (\sigma, \beta, A)$ consists of a stack $\sigma$, a buffer $\beta$, and a set $A$ of dependency arcs.[1] Given a sentence $W = w_1, \ldots, w_n$, the system is initialized with an empty stack, an empty arc set, and $\beta = w_1, \ldots, w_n, \textsc{root}$, where ROOT is the special root node. Any configuration $c$ with an empty stack and a buffer containing only ROOT is terminal, and the parse tree is given by the arc set $A_c$ of $c$.[2] The system has 4 transitions: $\textsc{Right}_{lb}$, $\textsc{Left}_{lb}$, SHIFT, REDUCE, defined as follows:

$$\textsc{Shift}[(\sigma,\ b|\beta,\ A)] \ = \ (\sigma|b,\ \beta,\ A)$$
$$\textsc{Right}_{lb}[(\sigma|s,\ b|\beta,\ A)] \ = \ (\sigma|s|b,\ \beta,\ A \cup \{(s, lb, b)\})$$
$$\textsc{Left}_{lb}[(\sigma|s,\ b|\beta,\ A)] \ = \ (\sigma,\ b|\beta,\ A \cup \{(b, lb, s)\})$$
$$\textsc{Reduce}[(\sigma|s,\ \beta,\ A)] = (\sigma,\ \beta,\ A)$$

[1] We use $\sigma|x$ to denote a stack with top element $x$ and remainder $\sigma$, and $x|\beta$ to denote a buffer with a head $x$ followed by the elements in $\beta$.

[2] This definition of a terminal configuration differs from that in Nivre (2003) but guarantees that the set $A_c$ is a dependency tree rooted in ROOT.

There is a precondition on the RIGHT and SHIFT transitions to be legal only when $b \neq$ ROOT, and for LEFT, RIGHT and REDUCE to be legal only when the stack is non-empty. Moreover, LEFT is only legal when $s$ does not have a parent in $A$, and REDUCE when $s$ does have a parent in $A$. In general, we use LEGAL$(c)$ to refer to the set of transitions that are legal in a configuration $c$. The arc-eager system builds trees *eagerly* in the sense that arcs are added at the earliest time possible. In addition, each word will collect all of its left dependents before collecting its right dependents.

**Arc-Standard** The arc-standard system (Nivre, 2004) has configurations of the same form $c = (\sigma, \beta, A)$ as the arc-eager system. The initial configuration for a sentence $W = w_1, \ldots, w_n$ has an empty stack and arc set and $\beta = \textsc{root}, w_1, \ldots, w_n$. A configuration $c$ is terminal if it has an empty buffer and a stack containing the single node ROOT; the parse tree is given by $A_c$. The system has 3 transitions: RIGHT$_{lb}$, LEFT$_{lb}$, SHIFT, defined as follows:

$$\textsc{Shift}[(\sigma,\ b|\beta,\ A)] = (\sigma|b,\ \beta,\ A)$$
$$\textsc{Right}_{lb}[(\sigma|s_1|s_0,\ \beta,\ A)] = (\sigma|s_1,\ \beta,\ A \cup \{(s_1, lb, s_0)\})$$
$$\textsc{Left}_{lb}[(\sigma|s_1|s_0,\ \beta,\ A)] = (\sigma|s_0,\ \beta,\ A \cup \{(s_0, lb, s_1)\})$$

There is a precondition on the LEFT transition to be legal only when $s_1 \neq$ ROOT, and for LEFT and RIGHT to be legal only when the stack has at least two elements. The arc-standard system builds trees in a *bottom-up* fashion: each word must collect all its dependents before being attached to its head. The system does not pose any restriction with regard to the order of collecting left and right dependents.

**Hybrid** The hybrid system (Kuhlmann et al., 2011) has the same configurations and the same initialization and termination conditions as the arc-standard system. The system has 3 transitions: RIGHT$_{lb}$, LEFT$_{lb}$, SHIFT, defined as follows:

$$\textsc{Shift}[(\sigma,\ b|\beta,\ A)] = (\sigma|b,\ \beta,\ A)$$
$$\textsc{Right}_{lb}[(\sigma|s_1|s_0,\ \beta,\ A)] = (\sigma|s_1,\ \beta,\ A \cup \{(s_1, lb, s_0)\})$$
$$\textsc{Left}_{lb}[(\sigma|s,\ b|\beta,\ A)] = (\sigma,\ b|\beta,\ A \cup \{(b, lb, s)\})$$

There is a precondition on RIGHT to be legal only when the stack has at least two elements, and on LEFT to be legal only when the stack is non-empty and $s \neq$ ROOT. The hybrid system can be seen as a combination of the arc-standard and arc-eager systems, using the LEFT action of arc-eager and the RIGHT action of arc-standard. Like arc-standard, it builds trees in a bottom-up fashion. But like arc-eager, it requires a word to collect all its left dependents before collecting any right dependent.

**Algorithm 1** Greedy transition-based parsing

```
Input: sentence $W$, parameter-vector $\mathbf{w}$
$c \leftarrow \textsc{Initial}(W)$
while not $\textsc{Terminal}(c)$ do
    $t_p \leftarrow \arg\max_{t \in \textsc{Legal}(c)} \mathbf{w} \cdot \phi(c, t)$
    $c \leftarrow t_p(c)$
return $A_c$
```

**Easy-First** In the easy-first system (Goldberg and Elhadad, 2010), a configuration $c = (\lambda, A)$ consists of a list $\lambda$ and a set $A$ of dependency arcs. We use $l_i$ to denote the $i$th member of $\lambda$ and write $|\lambda|$ for the length of $\lambda$. Given a sentence $W = w_1, \ldots, w_n$, the system is initialized with an empty arc set and $\lambda = \textsc{root}, w_1, \ldots, w_n$. A configuration $c$ is terminal with parse tree $A_c$ if $\lambda = \textsc{root}$. The set of transitions for a given configuration $c = (\lambda, A)$ is:

$$\{\textsc{Left}^i_{lb} | 1 < i \leq |\lambda|\} \cup \{\textsc{Right}^i_{lb} | 1 \leq i < |\lambda|\}, \text{ where:}$$
$$\textsc{Left}^i_{lb}[(\lambda, A)] = (\lambda \setminus \{l_{i-1}\},\ A \cup \{(l_i, lb, l_{i-1})\})$$
$$\textsc{Right}^i_{lb}[(\lambda, A)] = (\lambda \setminus \{l_{i+1}\},\ A \cup \{(l_i, lb, l_{i+1})\})$$

There is a precondition on LEFT$^i$ transitions to only trigger if $l_{i-1} \neq$ ROOT. Unlike the arc-eager, arc-standard and hybrid transition systems that work in a *left-to-right* order and access the sentence incrementally, the easy-first system is *non-directional* and has access to the entire sentence at each step. Like the arc-standard and hybrid systems, it builds trees bottom-up.

### 2.3 Greedy Transition-Based Parsing

Assuming that we have a feature-extraction function $\phi(c, t)$ over configurations $c$ and transitions $t$ and a weight-vector $\mathbf{w}$ assigning weights to each feature, greedy transition-based parsing is very simple and efficient using Algorithm 1. Starting in the initial configuration for a given sentence, we repeatedly choose the highest-scoring transition according to our model and apply it, until we reach a terminal configuration, at which point we stop and return the parse tree accumulated in the configuration.

**Algorithm 2** Online training of greedy transition-based parsers ($i$th iteration)

```
1:  for sentence $W$ with gold tree $T$ in corpus do
2:      $c \leftarrow \textsc{Initial}(W)$
3:      while not $\textsc{Terminal}(c)$ do
4:          $\textsc{Correct}(c) \leftarrow \{t \mid o(t; c, T) = \mathbf{true}\}$
5:          $t_p \leftarrow \arg\max_{t \in \textsc{Legal}(c)} \mathbf{w} \cdot \phi(c, t)$
6:          $t_o \leftarrow \arg\max_{t \in \textsc{Correct}(c)} \mathbf{w} \cdot \phi(c, t)$
7:          if $t_p \notin \textsc{Correct}(c)$ then
8:              $\textsc{Update}(\mathbf{w}, \phi(c, t_o), \phi(c, t_p))$
9:              $c \leftarrow \textsc{Next}(c, t_o)$
10:         else
11:             $c \leftarrow t_p(c)$
```

## 3 Training Transition-Based Parsers

We now turn to the training of greedy transition-based parsers, starting with a review of the standard method using static oracles and moving on to the idea of training with exploration proposed by Goldberg and Nivre (2012).

### 3.1 Training with Static Oracles

The standard approach to training greedy transition-based parsers is illustrated in Algorithm 2.[3] It assumes the existence of an *oracle* $o(t; c, T)$, which returns **true** if transition $t$ is correct for configuration $c$ and gold tree $T$. Given this oracle, training is very similar to parsing, but after predicting the next transition $t_p$ using the model in line 5 we check if it is contained in the set $\textsc{Correct}(c)$ of transitions that are considered correct by the oracle (lines 4 and 7). If the predicted transition $t_p$ is not correct, we update the model parameters $\mathbf{w}$ away from $t_p$ and toward the oracle prediction $t_o$, which is the highest-scoring correct transition under the current model, and move on to the next configuration (lines 7–9). If $t_p$ is correct, we simply apply it and move to $t_p(c)$ without changing the model parameters (line 11).

The function $\textsc{Next}(c, t_o)$ in line 9 is used to abstract over a subtle difference in the standard training procedure for the left-to-right systems (arc-eager, arc-standard and hybrid), on the one hand, and the easy-first system, on the other. In the former case, $\textsc{Next}(c, t_o)$ evaluates to $t_o(c)$, which means that we apply the oracle transition $t_o$ and move on to the next configuration. For the easy-first system, $\textsc{Next}(c, t_o)$ instead evaluates to $c$, which means that we remain in the same configuration for as many updates as necessary to get a correct model prediction.

Traditionally, the oracles for the left-to-right systems are *static*: they return a single correct transition and are only correct for configurations that result from transitions predicted by the oracle itself. The oracle for the easy-first system is *non-deterministic* and returns a set of correct transitions. However, like the static oracle, it is correct only for configurations from which the gold tree is reachable. Thus, in both cases, we need to make sure that a transition is applied during training only if it is considered correct by the oracle; else we cannot guarantee that later oracle predictions will be correct. Therefore, on line 9, we either remain in the same configuration (easy-first) or follow the oracle prediction and go to $t_o(c)$ (left-to-right systems); on line 11, we in fact also go to $t_o(c)$, because in this case we have $t_p(c) = t_o(c)$.

A notable shortcoming of this training procedure is that, at parsing time, the parsing model may predict incorrect transitions and reach configurations that are not on the oracle path. Since the model has never seen such configurations during training, it is likely to perform badly in them, making further mistakes more likely. We would therefore like the parser to encounter configurations resulting from incorrect transitions during training and learn what constitutes optimal transitions in such configurations. Unfortunately, this is not possible using the static (or even the non-deterministic) oracles.

### 3.2 Training with Exploration

Assuming we had access to an oracle that could tell us which transitions are *optimal* in any configuration, including ones from which the gold tree is not reachable, we could trivially change the training algorithm to incorporate learning on configurations that result from incorrect transitions, and thereby mitigate the effects of error propagation at parsing time. Conceptually, all that we need to change is line 9. Instead of following the prediction $t_p$ only when it is correct (line 11), we could sometimes choose to follow $t_p$ also when it is not correct.

[3] We present the standard approach as an online algorithm in order to ease the transition to the novel approach. While some transition-based parsers use batch learning instead, the essential point is that they explore exactly the same configurations during the training phase.

**Algorithm 3** Online training with exploration for greedy transition-based parsers ($i$th iteration)

```
1:  for sentence $W$ with gold tree $T$ in corpus do
2:      $c \leftarrow \textsc{Initial}(W)$
3:      while not $\textsc{Terminal}(c)$ do
4:          $\textsc{Correct}(c) \leftarrow \{t | o(t; c, T) = \mathbf{true}\}$
5:          $t_p \leftarrow \arg\max_{t \in \textsc{Legal}(c)} \mathbf{w} \cdot \phi(c, t)$
6:          $t_o \leftarrow \arg\max_{t \in \textsc{Correct}(c)} \mathbf{w} \cdot \phi(c, t)$
7:          if $t_p \notin \textsc{Correct}(c)$ then
8:              $\textsc{Update}(\mathbf{w}, \phi(c, t_o), \phi(c, t_p))$
9:              $c \leftarrow \textsc{Explore}_{k,p}(c, t_o, t_p, i)$
10:         else
11:             $c \leftarrow t_p(c)$

1:  function $\textsc{Explore}_{k,p}(c, t_o, t_p, i)$
2:      if $i > k$ and $\textsc{Rand}() < p$ then
3:          return $t_p(c)$
4:      else
5:          return $\textsc{Next}(c, t_o)$
```

The rest of the training algorithm does not need to change, as the set $\textsc{Correct}(c)$ obtained in line 4 would now include the set of optimal transitions to take from configurations reached by following the incorrect transition, as provided by the new oracle. Following Goldberg and Nivre (2012), we call this approach *learning with exploration*. The modified training procedure is specified in Algorithm 3.

There are three major questions that need to be answered when implementing a concrete version of this algorithm:

**Exploration Policy** When do we follow an incorrect transition, and which one do we follow?

**Optimality** What constitutes an *optimal* transition in configurations from which the gold tree is not reachable?

**Oracle** Given a definition of optimality, how do we calculate the set of optimal transitions in a given configuration?

The first two questions are independent of the specific transition system. In our experiments, we use a simple exploration policy, parameterized by an iteration number $k$ and a probability $p$. This policy always chooses an oracle transition during the first $k$ iterations but later chooses the oracle transition with probability $1 - p$ and the (possibly incorrect) model prediction otherwise. This is defined in the function $\textsc{Explore}_{k,p}(c, t_o, t_p, i)$ (called in line 9 of Algorithm 3), which takes two additional arguments compared to Algorithm 2: the model prediction $t_p$ and the current training iteration $i$. If $i$ exceeds the iteration threshold $k$ and if a randomly generated probability does not exceed the probability threshold $p$, then the function returns $t_p(c)$, which means that we follow the (incorrect) model prediction. Otherwise, it reverts to the old $\textsc{Next}(c, t_o)$ function, returning $c$ for easy-first and $t_o(c)$ for the other systems. We show in Section 5 that the training procedure is relatively insensitive to the choice of $k$ and $p$ values as long as predicted transitions are chosen often.

Our optimality criterion is directly related to the attachment score metrics commonly used to evaluate dependency parsers.[4] We say that a transition $t$ is optimal in a configuration $c$ if and only if the best achievable attachment score from $t(c)$ is equal to the best achievable attachment score from $c$.

The implementation of oracles is specific to each transition system. In the next section, we first provide a characterization of *complete non-deterministic oracles*, also called *dynamic oracles*, which is what we require for the training procedure in Algorithm 3. We then define a property of transition systems which we call *arc decomposition* and present a general method for deriving complete non-deterministic oracles for arc-decomposable systems. Finally, we use this method to derive concrete oracles for the arc-eager, hybrid and easy-first systems, which are all arc-decomposable. In Section 5, we then show experimentally that we indeed achieve better parsing accuracy when using exploration during training.

## 4 Oracles for Transition-Based Parsing

Almost all greedy transition-based parsers described in the literature are trained using what we call *static* oracles. We now make this notion precise and contrast it with *non-deterministic* and *complete* oracles. Following the terminology of Goldberg and Nivre

[4] The labeled attachment score (LAS) is the percentage of words in a sentence that are assigned both the correct head and the correct label. The unlabeled attachment score (UAS) is the percentage of words that are assigned the correct head (regardless of label).

(2012), we reserve the term *dynamic oracles* for oracles that are both non-deterministic and complete.

### 4.1 Characterizing Oracles

During training, we assume that the oracle is a boolean function $o(t; c, T)$, which returns **true** if and only if transition $t$ is correct in configuration $c$ for gold tree $T$ (cf. Algorithms 2–3). However, such a function may be defined in terms of different underlying functions that we also call oracles.

A *static* oracle is a function $o_s(T)$ mapping a tree $T$ to a sequence of transitions $t_1, \ldots, t_n$. A static oracle is correct if starting in the initial configuration and applying the transitions in $o_s(T)$ in order results in the transition system reaching a terminating configuration with parse tree $T$. Formally, a static oracle is correct if and only if, for every projective dependency tree $T$ with yield $W$, $o_s(T) = t_1, \ldots, t_n$, $c = t_n(\ldots(t_1(\textsc{Initial}(W))))$, $\textsc{Terminal}(c)$ and $A_c = T$.[5] When using a static oracle for training in Algorithm 2, the function $o(t; c, T)$ returns **true** if $o_s(T) = t_1, \ldots, t_n$, $c = t_{i-1}(\ldots(t_1(\textsc{Initial}(W))))$ (for some $i$, $1 \leq i \leq n$) and $t = t_i$. If $t \neq t_i$, $o(t; c, T) =$ **false**; if $c \neq t_{i-1}(\ldots(t_1(\textsc{Initial}(W))))$ (for all $i$, $1 \leq i \leq n$), $o(t; c, T)$ is undefined. A static oracle is therefore essentially incomplete, because it is only defined for configurations that are part of the oracle path.[6] Static oracles either allow a single transition at a given configuration, or are undefined for that configuration.

By contrast, a *non-deterministic* oracle is a function $o_n(c, T)$ mapping a configuration $c$ and a tree $T$ to a *set* of transitions. A non-deterministic oracle is correct if and only if, for every projective dependency tree $T$, every configuration $c$ from which $T$ is reachable, and every transition $t \in o_n(c, T)$, $t(c)$ is a configuration from which $T$ is still reachable. Note that this definition of correctness for non-deterministic oracles is restricted to configurations from which a goal tree is reachable. Non-deterministic oracles are more flexible than static oracles in that they allow for spurious ambiguity: they support the possibility of different sequences of transitions leading to the gold tree. However, they are still only guaranteed to be correct on a subset of the possible configurations. Thus, when using a non-deterministic oracle for training in Algorithm 2, the function $o(t; c, T)$ returns **true** if $T$ is reachable from $c$ and $t \in o_n(c, T)$. However, if $T$ is not reachable from $c$, $o(t; c, T)$ is not necessarily well-defined.

A *complete* non-deterministic oracle is a function $o_d(c, T)$ for which this restriction is removed, so that correctness is defined over all configurations that are reachable from the initial configuration. Following Goldberg and Nivre (2012), we call complete non-deterministic oracles *dynamic*. In order to define correctness for dynamic oracles, we must first introduce a *cost function* $\mathcal{C}(A, T)$, which measures the cost of outputting parse $A$ when the gold tree is $T$. In this paper, we define cost as Hamming loss (for labeled or unlabeled dependency arcs), which is directly related to the attachment score metrics used to evaluate dependency parsers, but other cost functions are conceivable. We say that a complete non-deterministic oracle is correct if and only if, for every projective dependency tree $T$ with yield $W$, every configuration $c$ that is reachable from $\textsc{Initial}(W)$, and every transition $t \in o_d(c, T)$, $\min_{A: c \rightsquigarrow A} \mathcal{C}(A, T) = \min_{A: t(c) \rightsquigarrow A} \mathcal{C}(A, T)$, where $c \rightsquigarrow A$ signifies that the parse $A$ is reachable from $c$, a notion that will be formally defined in the next subsection. In other words, even if the gold tree $T$ is no longer reachable itself, the best tree reachable from $t(c)$ has the same cost as the best tree reachable from $c$.

In addition to a cost function for arc sets and trees, it is convenient to define a cost function for transitions. We define $\mathcal{C}(t; c, T)$ to be the difference in cost between the best tree reachable from $t(c)$ and $c$, respectively. That is:

$$\mathcal{C}(t; c, T) = \min_{A: t(c) \rightsquigarrow A} \mathcal{C}(A, T) - \min_{A: c \rightsquigarrow A} \mathcal{C}(A, T)$$

A dynamic oracle can then be defined as an oracle that returns the set of transitions with zero cost:

$$o_d(c, T) = \{t \mid \mathcal{C}(t; C, T) = 0\}$$

[5] Since all the transition systems considered in this paper are restricted to projective dependency trees, we only define correctness with respect to this class. There are obvious generalizations that apply to more expressive transition systems.

[6] Static oracles are usually described as rules over parser configurations, i.e., "if the configuration is X take transition Y", giving the impressions they are functions from configurations to transitions. However, as explained here, these rules are only correct if the sequence of transitions is followed in its entirety.

### 4.2 Arc Reachability and Arc Decomposition

We now define the notion of reachability for parses (or arc sets), used already in the previous subsection, and relate it to reachability for individual dependency arcs. This enables us to define a property of transition systems called arc decomposition, which is very useful when deriving dynamic oracles.

**Arc Reachability** We say that a dependency arc $(h, d)$[7] is *reachable* from a configuration $c$, written $c \rightsquigarrow (h, d)$, if there is a (possibly empty) sequence of transitions $t_1, \ldots, t_k$ such that $(h, d) \in A_{(t_k(\ldots t_1(c)))}$. In words, we require a sequence of transitions starting from $c$ and leading to a configuration whose arc set contains $(h, d)$.

**Arc Set Reachability** A set of dependency arcs $A = \{(h_1, d_1), \ldots, (h_n, d_n)\}$ is *reachable* from a configuration $c$, written $c \rightsquigarrow A$, if there is a (possibly empty) sequence of transitions $t_1, \ldots, t_k$ such that $A \subseteq A_{(t_k(\ldots t_1(c)))}$. In words, there is a sequence of transitions starting from $c$ and leading to a configuration where all arcs in $A$ have been derived.

**Tree Consistency** A set of arcs $A$ is said to be *tree consistent* if there exists a projective dependency tree $T$ such that $A \subseteq T$.

**Arc Decomposition** A transition system is said to be *arc decomposable* if, for every tree consistent arc set $A$ and configuration $c$, $c \rightsquigarrow A$ is entailed by $c \rightsquigarrow (h, d)$ for every arc $(h, d) \in A$. In words, if every arc in a tree consistent arc set is reachable from a configuration, then the entire arc set is also reachable from that configuration.

Arc decomposition is a powerful property, allowing us to reduce reasoning about the reachability of arc sets or trees to reasoning about the reachability of individual arcs, and will later use this property to derive dynamic oracles for the arc-eager, hybrid and easy-first systems.

### 4.3 Proving Arc Decomposition

Let us now sketch how arc decomposition can be proven for the transition systems in consideration.

**Arc-Eager** For the arc-eager system, consider an arbitrary configuration $c = (\sigma, \beta, A)$ and a tree-consistent arc set $A'$ such that all arcs are reachable from $c$. We can partition $A'$ into four sets, each of which is by necessity itself a tree-consistent arc-set:

(1) $\overline{\mathcal{B}} = \{(h, d) \mid h, d \notin \beta\}$
(2) $\mathcal{B} = \{(h, d) \mid h, d \in \beta\}$
(3) $\mathcal{B}_h = \{(h, d) \mid h \in \beta, d \in \sigma\}$
(4) $\mathcal{B}_d = \{(h, d) \mid d \in \beta, h \in \sigma\}$

Arcs in $\overline{\mathcal{B}}$ are already in $A$ and cannot interfere with other arcs. $\mathcal{B}$ is reachable by any sequence of transitions that derives a tree consistent with $\mathcal{B}$ for a sentence containing only the words in $\beta$. In deriving this tree, every node $x$ involved in some arc in $\mathcal{B}_h$ or $\mathcal{B}_d$ must at least once be at the head of the buffer. Let $c_x$ be the first such configuaration. From $c_x$, every arc $(x, d) \in \mathcal{B}_h$ can be derived without interfering with arcs in $A'$ by a sequence of REDUCE and LEFT-ARC$_{lb}$ transitions. This sequence of transitions will trivially not interfere with other arcs in $\mathcal{B}_h$. Moreover, it will not interfere with arcs in $\mathcal{B}_d$ because $A'$ is tree consistent and projectivity ensures that an arc of the form $(y, z)$ $(y \in \sigma, z \in \beta)$ must satisfy $y < d < x \leq z$. Finally, it will not interfere with arcs in $\mathcal{B}$ because the buffer remains unchanged. After deriving every arc $(x, d) \in \mathcal{B}_h$, we remain with at most one $(h, x) \in \mathcal{B}_d$ (because of the single-head constraint). By the same reasoning as above, a sequence of REDUCE and LEFT-ARC$_{lb}$ transitions will take us to a configuration where $h$ is on top of the stack without interfering with arcs in $A'$. We can then derive the arc $(h, x)$ using RIGHT-ARC$_{lb}$. This does not interfere with arcs remaining in $\mathcal{B}_h$ or $\mathcal{B}_d$ because all such arcs must have their buffer node further down the buffer (due to projectivity). At this point, we have reached a configuration $c_{x+1}$ to which the same reasoning applies for the next node $x + 1$.

**Hybrid** The proof for the hybrid system is very similar but with a slightly different partitioning because of the bottom-up order and the different way of handling right-arcs.

[7] We consider unlabeled arcs here in order to keep notation simple. Everything is trivially extendable to the labeled case.

**Easy-First** For the easy-first system, we only need to partition arcs into $\overline{\mathcal{L}} = \{(h,d) \,|\, d \notin \lambda\}$ and $\mathcal{L} = \{(h,d) \,|\, h,d \in \lambda\}$. The former must already be in $A$, and for the latter there can be no conflict between arcs as long as we respect the bottom-up ordering.

**Arc-Standard** Unfortunately, arc decomposition does *not* hold for the arc-standard system. To see why, consider a configuration with the stack $\sigma = a, b, c$. The arc $(c, b)$ is reachable via LEFT, the arc $(b, a)$ is reachable via RIGHT, LEFT, the arc set $A = \{(c, b), (b, a)\}$ forms a projective tree and is thus tree consistent, but it is easy to convince oneself that $A$ is not reachable from this configuration. The reason that the above proof technique fails for the arc-standard system is that the arc set corresponding to $\mathcal{B}$ in the arc-eager system may involve arcs where both nodes are still on the stack, and we cannot guarantee that all projective trees consistent with these arcs can be derived. In the very similar hybrid system, such arcs exist as well but they are limited to arcs of the form $(h, d)$ where $h < d$ and $h$ and $d$ are adjacent on the stack, and this restriction is sufficient to restore arc decomposition.

### 4.4 Deriving Oracles

We now present a procedure for deriving a dynamic oracle for any arc-decomposable system. First of all, we can define a non-deterministic oracle as follows:

$$o_n(c, T) = \{t \mid t(c) \rightsquigarrow T\}$$

That is, we allow all transitions after which the goal tree is still reachable. Note that if $c \rightsquigarrow T$ holds, then the set returned by the oracle is guaranteed to be non-empty. For a sound and complete transition system, we know that INITIAL$(W) \rightsquigarrow T$ for any projective dependency tree with yield $W$, and the oracle is guaranteed to return a non-empty set as long as we are not in the terminal configuration and have followed transitions suggested by the oracle.

In order to extend the non-deterministic oracle to a dynamic oracle, we make use of the transition cost function introduced earlier:

$$o_d(c, T) = \{t \mid \mathcal{C}(t; c, T) = 0\}$$

As already mentioned, we assume here that the cost is the difference in Hamming loss between the best tree reachable before and after the transition.[8] Assuming arc decomposition, this is equivalent to the number of gold arcs that are reachable before but not after the transition. For configurations from which $T$ is reachable, the dynamic oracle coincides with the non-deterministic oracle. But for configurations from which $T$ cannot be derived, the dynamic oracle returns transitions leading to the best parse $A$ (in terms of Hamming distance from $T$) which is reachable from $c$. This is the behavior expected from a dynamic oracle, as defined in Section 4.1.

Thus, in order to derive a dynamic oracle for an arc-decomposable transition system, it is sufficient to show that the transition cost function $C(t; c, T)$ can be computed efficiently for that system.[9] Next we show how to do this for the arc-eager, hybrid and easy-first systems.

### 4.5 Concrete Oracles

In a given transition system, the set of individually reachable arcs is relatively straightforward to compute. In an arc-decomposable system, we know that any intersection of the set of individually reachable arcs with a projective tree is tree consistent, and therefore also reachable. In particular, this holds for the goal tree. For such systems, we can therefore compute the transition cost by intersecting the set of arcs that are individually reachable from a configuration with the goal arc set, and see how a given transition affects this set of reachable arcs.

**Arc-Eager** In the arc-eager system, an arc $(h, d)$ is reachable from a configuration $c$ if one of the following conditions hold:

(1) $(h, d)$ is already derived ($(h, d) \in A_c$);
(2) $h$ and $d$ are in the buffer;
(3) $h$ is on the stack and $d$ is in the buffer;
(4) $d$ is on the stack and is not assigned a head and $h$ is in the buffer.

[8] The framework is easily adapted to a different cost function such as weighted Hamming cost, where different gold arcs are weighted differently.

[9] In fact, in order to use the dynamic oracle with our current learning algorithm, we do not need the full power of the *cost* function: it is sufficient to distinguish between transitions with zero cost and transitions with non-zero cost.

The cost function for a configuration of the form $c = (\sigma|s, b|\beta, A)$[10] can be calculated as follows:[11]

- $\mathcal{C}(\text{LEFT}; c, T)$: Adding the arc $(b, s)$ and popping $s$ from the stack means that $s$ will not be able to acquire any head or dependents in $\beta$. The cost is therefore the number of arcs in $T$ of the form $(k, s)$ or $(s, k)$ such that $k \in \beta$. Note that the cost is 0 for the trivial case where $(b, s) \in T$, but also for the case where $b$ is not the gold head of $s$ but the real head is not in $\beta$ (due to an erroneous previous transition) and there are no gold dependents of $s$ in $\beta$.

- $\mathcal{C}(\text{RIGHT}; c, T)$: Adding the arc $(s, b)$ and pushing $b$ onto the stack means that $b$ will not be able to acquire any head in $\sigma$ or $\beta$, nor any dependents in $\sigma$. The cost is therefore the number of arcs in $T$ of the form $(k, b)$, such that $k \in \sigma \cup \beta$, or of the form $(b, k)$ such that $k \in \sigma$ and there is no arc $(x, k)$ in $A_c$. Note again that the cost is 0 for the trivial case where $(s, b) \in T$, but also for the case where $s$ is not the gold head of $b$ but the real head is not in $\sigma$ or $\beta$ (due to an erroneous previous transition) and there are no gold dependents of $b$ in $\sigma$.

- $\mathcal{C}(\text{REDUCE}; c, T)$: Popping $s$ from the stack means that $s$ will not be able to acquire any dependents in $B = b|\beta$. The cost is therefore the number of arcs in $T$ of the form $(s, k)$ such that $k \in B$. While it may seem that a gold arc of the form $(k, s)$ should be accounted for as well, note that a gold arc of that form, if it exists, is already accounted for by a previous (erroneous) RIGHT transition when $s$ acquired its head.

- $\mathcal{C}(\text{SHIFT}; c, T)$: Pushing $b$ onto the stack means that $b$ will not be able to acquire any head or dependents in $S = s|\sigma$. The cost is therefore the number of arcs in $T$ of the form $(k, b)$ or $(b, k)$ such that $k \in S$ and (for the second case) there is no arc $(x, k)$ in $A_c$.

**Hybrid** In the hybrid system, an arc $(h, d)$ is reachable from a configuration $c$ if one of the following conditions holds:

(1) $(h, d)$ is already derived ($(h, d) \in A_c$);
(2) $h$ and $d$ are in the buffer;
(3) $h$ is on the stack and $d$ is in the buffer;
(4) $d$ is on the stack and $h$ is in the buffer;
(5) $d$ is in stack location $i$, $h$ is in stack location $i - 1$ (that is, the stack has the form $\sigma = \ldots, h, d, \ldots$).

The cost function for a configuration of the form $c = (\sigma|s_1|s_0, b|\beta, A)$[12] can be calculated as follows:

- $\mathcal{C}(\text{LEFT}; c, T)$: Adding the arc $(b, s_0)$ and popping $s_0$ from the stack means that $s_0$ will not be able to acquire heads from $H = \{s_1\} \cup \beta$ and will not be able to acquire dependents from $D = \{b\} \cup \beta$. The cost is therefore the number of arcs in $T$ of the form $(s_0, d)$ and $(h, s_0)$ for $h \in H$ and $d \in D$.

- $\mathcal{C}(\text{RIGHT}; c, T)$: Adding the arc $(s_1, s_0)$ and popping $s_0$ from the stack means that $s_0$ will not be able to acquire heads or dependents from $B = \{b\} \cup \beta$. The cost is therefore the number of arcs in $T$ of the form $(s_0, d)$ and $(h, s_0)$ for $h, d \in B$.

- $\mathcal{C}(\text{SHIFT}; c, T)$: Pushing $b$ onto the stack means that $b$ will not be able to acquire heads from $H = \{s_1\} \cup \sigma$, and will not be able to acquire dependents from $D = \{s_0, s_1\} \cup \sigma$. The cost is therefore the number of arcs in $T$ of the form $(b, d)$ and $(h, b)$ for $h \in H$ and $d \in D$.

**Easy-First** In the easy-first system, an arc $(h, d)$ is reachable from a configuration $c$ if one of the following conditions holds:

(1) $(h, d)$ is already derived ($(h, d) \in A_c$);
(2) $h$ and $d$ are in the list $\lambda$.

When adding an arc $(h, d)$, $d$ is removed from the list $\lambda$ and cannot participate in any future arcs. Thus, a transition has a cost $> 0$ with respect to a tree $T$ if one of the following holds:

[10] This is a slight abuse of notation, since for the SHIFT transition $s$ may not exist, and for the REDUCE transition $b$ may not exist.

[11] While very similar to the presentation in Goldberg and Nivre (2012), this version includes a small correction to the RIGHT and SHIFT transitions.

[12] Note again that $s_0$ may be missing in the case of SHIFT case and $s_1$ in the case of SHIFT and LEFT.

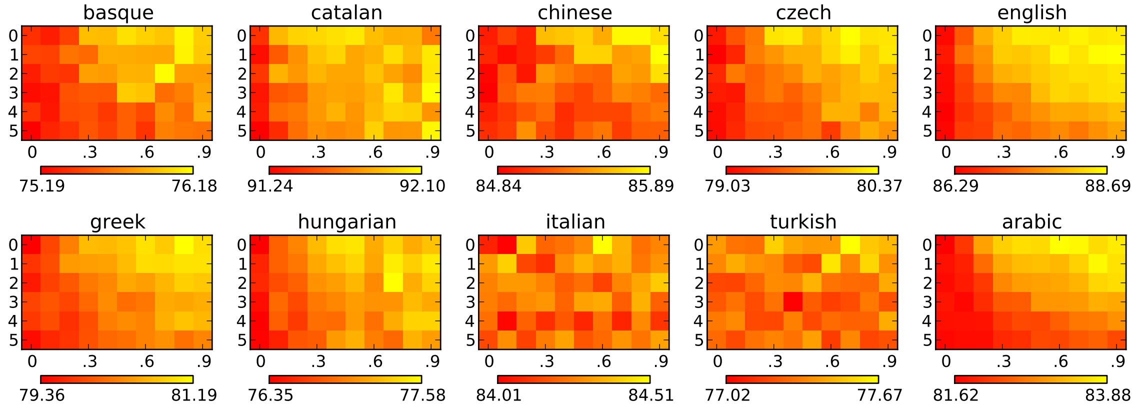

Figure 1: Effect of $k$ (y axis) and $p$ (x axis) values on parsing accuracies for the arc-eager system on the various CoNLL-2007 shared-task languages. Each point is an average UAS of 5 runs with different seeds. The general trend is that smaller $k$ and higher $p$ are better.

(1) it adds an arc $(h, d)$ such that $(h', d) \in T$ for some $h' \in \lambda$, $h' \neq h$;

(2) it adds an arc $(h, d)$ such that $(d, d') \in T$ for some $d' \in \lambda$.

The exact cost can be calculated by counting the number of such arcs.

## 5 Experiments and Results

**Setup, data and parameters** The goal of our experiments is to evaluate the utility of the dynamic oracles for training, by comparing a training scenario which only sees configurations that can lead to the gold tree (following a static oracle for the left-to-right systems and a non-deterministic but incomplete oracle for the easy-first system), against a training scenario that involves exploration of incorrect states, using the dynamic oracles.

As our training algorithm involves a random component (we shuffle the sentences prior to each iteration, and randomly select whether to follow a correct or incorrect action), we evaluate each setup five times using different random seeds, and report the averaged results.

We perform all of the experiments on the multilingual CoNLL-2007 data sets. We use 15 training iterations for the left-to-right parsers, and 20 training iterations for the easy-first parser. We use the standard perceptron update as our update rule in training, and use the averaged weight vector for prediction in test time. The feature sets differ by transition system but are kept the same across data sets. The exact feature-set definitions for the different systems are available in the accompanying software, which is available on line at the first author's homepage.

**Effect of exploration parameters** In an initial set of experiments, we investigate the effect of the exploration parameters $k$ and $p$ on the arc-eager system. The results are presented in Figure 1. While the optimal parameters vary by data set, there is a clear trend toward lower values of $k$ and higher values of $p$. This is consistent with the report of Goldberg and Nivre (2012) who used a fixed small value of $k$ and large value of $p$ throughout their experiments.

**Training with exploration for the various systems** For the second experiment, in which we compared training with a static oracle to training with exploration, we fixed the exploration parameters to $k = 1$ and $p = 0.9$ for all data sets and transition-system combinations. The results in terms of labeled accuracies (for the left-to-right systems) and unlabeled accuracies (for all systems) are presented in Table 1. Training with exploration using the dynamic oracles yields improved accuracy for the vast majority of the setups. The notable exceptions are the arc-eager and easy-first systems for unlabeled Italian and the arc-hybrid system in Catalan, where we observe a small drop in accuracy. However, we can safely conclude that training with exploration is beneficial and note that we may get even further gains in the future using better methods for tuning the exploration parameters or better training methods.

| system / language | hungarian | chinese | greek | czech | basque | catalan | english | turkish | arabic | italian |
|---|---|---|---|---|---|---|---|---|---|---|
| | UAS | | | | | | | | | |
| eager:static | 76.42 | 85.01 | 79.53 | 78.70 | 75.14 | 91.30 | 86.10 | 77.38 | 81.59 | 84.40 |
| eager:dynamic | 77.48 | 85.89 | 80.98 | 80.25 | 75.97 | 92.02 | 88.69 | 77.39 | 83.62 | 84.30 |
| hybrid:static | 76.39 | 84.96 | 79.40 | 79.71 | 73.18 | 91.30 | 86.43 | 75.91 | 83.43 | 83.43 |
| hybrid:dynamic | 77.54 | 85.10 | 80.49 | 80.07 | 73.70 | 91.06 | 87.62 | 76.90 | 84.04 | 83.83 |
| easyfirst:static | 81.27 | 87.01 | 81.28 | 82.00 | 75.01 | 92.50 | 88.57 | 78.92 | 82.73 | 85.31 |
| easyfirst:dynamic | 81.52 | 87.48 | 82.25 | 82.39 | 75.87 | 92.85 | 89.41 | 79.29 | 83.70 | 85.11 |
| | LAS | | | | | | | | | |
| eager:static | 66.72 | 81.24 | 72.44 | 71.08 | 65.34 | 86.02 | 84.93 | 66.59 | 72.10 | 80.17 |
| eager:dynamic | 68.41 | 82.23 | 73.81 | 72.99 | 66.63 | 86.93 | 87.69 | 67.05 | 73.92 | 80.43 |
| hybrid:static | 66.54 | 80.17 | 70.99 | 71.88 | 62.84 | 85.57 | 84.96 | 64.80 | 73.16 | 78.78 |
| hybrid:dynamic | 68.05 | 80.59 | 72.07 | 72.15 | 63.52 | 85.47 | 86.28 | 66.12 | 74.10 | 79.25 |

Table 1: Results on the CoNLL 2007 data set. UAS, including punctuation. Each number is an average over 5 runs with different randomization seeds. All experiments used the same exploration parameters of $k$=1, $p$=0.9.

## 6 Related Work

The error propagation problem for greedy transition-based parsing was diagnosed by McDonald and Nivre (2007) and has been tackled with a variety of techniques including parser stacking (Nivre and McDonald, 2008; Martins et al., 2008) and beam search and structured prediction (Zhang and Clark, 2008; Zhang and Nivre, 2011). The technique called bootstrapping in Choi and Palmer (2011) is similar in spirit to training with exploration but is applied iteratively in batch mode and is only approximate due to the use of static oracles. Dynamic oracles were first explored by Goldberg and Nivre (2012).

In machine learning more generally, our approach can be seen as a problem-specific instance of *imitation learning* (Abbeel and Ng, 2004; Vlachos, 2012; He et al., 2012; Daumé III et al., 2009; Ross et al., 2011), where the dynamic oracle is used to implement the *optimal expert* needed in the imitation learning setup. Indeed, our training procedure is closely related to DAgger (Ross et al., 2011), which also trains a classifier to match an expert on a distribution of possibly suboptimal states obtained by running the system itself. Our training procedure can be viewed as an online version of DAgger (He et al., 2012) with two extensions: First, our learning algorithm involves a stochastic policy parameterized by $k$, $p$ for choosing between the oracle or the model prediction, whereas DAgger always follows the system's own prediction (essentially running with $k = 0$, $p = 1$). The heatmaps in Figure 1 show that this parameterization is beneficial. Second, while DAgger assumes an expert providing a single label at each state, our oracle is nondeterministic and allows multiple correct labels (transitions) which our training procedure tie-breaks according to the model's current prediction, a technique that has recently been proposed in an extension to DAgger by He et al. (2012). Other related approaches in the machine learning literature include stacked sequential learning (Cohen and Carvalho, 2005), LaSO (Daumé III and Marcu, 2005), Searn (Daumé III et al., 2009) and SMILe (Ross and Bagnell, 2010).

## 7 Conclusion

In this paper, we have extended the work on dynamic oracles presented in Goldberg and Nivre (2012) in several directions by giving formal characterizations of non-deterministic and complete oracles, defining the arc-decomposition property for transition systems, and using this property to derive novel complete non-deterministic oracles for the hybrid and easy-first systems (as well as a corrected oracle for the arc-eager system). We have then used the completeness of these new oracles to improve the training procedure of greedy parsers to include explorations of configurations which result from incorrect transitions. For all three transition systems, we get substantial accuracy improvements on many languages. As the changes all take place at training time, the very fast running time of the greedy algorithm at test time is maintained.

## References

Pieter Abbeel and Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, page 1.

Jinho D. Choi and Martha Palmer. 2011. Getting the most out of transition-based dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 687–692.

William W. Cohen and Vitor R. Carvalho. 2005. Stacked sequential learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 671–676.

Hal Daumé III and Daniel Marcu. 2005. Learning as search optimization: Approximate large margin methods for structured prediction. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 169–176.

Hal Daumé III, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine Learning*, 75:297–325.

Yoav Goldberg and Michael Elhadad. 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 (NAACL HLT)*, pages 742–750.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 959–976.

He He, Hal Daumé III, and Jason Eisner. 2012. Imitation learning by coaching. In *Advances in Neural Information Processing Systems 25*.

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

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–11.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 673–682.

André Filipe Martins, Dipanjan Das, Noah A. Smith, and Eric P. Xing. 2008. Stacking dependency parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 157–166.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 91–98.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 950–958.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160.

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

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:513–553.

Stéphane Ross and J. Andrew Bagnell. 2010. Efficient reductions for imitation learning. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pages 661–668.

Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 627–635.

Andreas Vlachos. 2012. An investigation of imitation learning algorithms for structured prediction. In *Proceedings of the European Workshop on Reinforcement Learning (EWRL)*, pages 143–154.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 562–571.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193.