Non-Deterministic Oracles for Unrestricted Non-Projective Transition-Based Dependency Parsing

Anders Björkelund
University of Stuttgart
Institute for Natural Language Processing
Stuttgart, Germany
anders@ims.uni-stuttgart.de

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

Abstract

We study non-deterministic oracles for training non-projective beam search parsers with swap transitions. We map out the spurious ambiguities of the transition system and present two non-deterministic oracles as well as a static oracle that minimizes the number of swaps. An evaluation on 10 treebanks reveals that the difference between static and non-deterministic oracles is generally insignificant for beam search parsers but that non-deterministic oracles can improve the accuracy of greedy parsers that use swap transitions.

1 Introduction

Training transition-based dependency parsers relies on an **oracle** – a function that, given a parser configuration and a gold dependency tree, returns the correct transition. The sequence of transitions required to derive a given dependency tree is, however, typically not unique, and for certain configurations more than one transition is correct. This issue has typically been dealt with by defining a canonical order among the transitions, thereby resolving such ambiguities in a deterministic way. In addition to the determinism, standard oracles also make the assumption that the gold tree can be recovered from the current configuration. Oracles with this behavior are known as **static** oracles.

Recently, much work has been devoted to the development of *dynamic* oracles that do away with both of these assumptions (Goldberg and Nivre, 2013; Goldberg et al., 2014; Gómez-Rodríguez et al., 2014). Dynamic oracles have been shown to be very successful for training greedy parsers. Since greedy parsers typically suffer from error propagation, dynamic oracles enable the parsers to learn to do “the next best thing” after having made a mistake, resulting in considerable improvements in parsing accuracy.

Nevertheless, greedy transition-based parsers still lag behind search-based parsers that explore a larger set of possible transition sequences. Search-based parsers are typically realized through beam search and trained using global learning, where a discriminative model is trained to score not just single transitions, but a sequence of transitions (Zhang and Clark, 2008). The combination of non-greedy inference and global learning enables search-based parsers to overcome the error propagation problem. However, since the model is globally trained, oracles that can recover from past mistakes and do the next best thing are not applicable in this scenario. On the other hand, it is an open question whether search-based parsers should be trained using static oracles, or whether their performance can be further increased by using a **non-deterministic** oracle, i.e., an oracle that considers all possible transition sequences that can derive the gold dependency tree.

We evaluate the hypothesis that transition-based parsers with beam search can be improved by using non-deterministic oracles during training. We do this in the context of the transition system by Nivre (2009), henceforth SwapStandard, which extends the ArcStandard system (Nivre, 2004) with a swap transition to accommodate non-projective dependency trees. This system has been shown to be very effective with beam search, even rivaling graph-based dependency parsers (Bohnet and Nivre, 2012; Bohnet et al., 2013). Empirically, we find that the utility of non-deterministic oracles for training beam search parsers is rather limited and typically the difference compared to a static oracle is insignificant. However, our experiments also show that the non-deterministic oracles can be beneficial when training greedy parsers, a result that has not previously been shown for non-projective systems.

The main contribution of this paper is the first characterization of a non-deterministic oracle for

Proceedings of the 14th International Conference on Parsing Technologies, pages 76–86,
Bilbao, Spain; July 22–24, 2015. ©2015 Association for Computational Linguistics

the SwapStandard system, based on a thorough analysis of spurious ambiguities. As a side-result, we also arrive at a static oracle that minimizes the number of swap transitions required to parse any non-projective dependency tree, thereby solving a previously open problem (Nivre, 2009; Nivre et al., 2009). In addition, we provide the first empirical evaluation of non-deterministic oracles for training beam search parsers, as well as the first evaluation with greedy parsers using a non-projective transition system.

2 Related Work

During the last decade, a plethora of transition systems has been described. Early systems, such as ArcEager (Nivre, 2003) and ArcStandard (Nivre, 2004) were restricted to projective structures. Several systems that can accommodate non-projective structures have subsequently been described (Attardi, 2006; Gómez-Rodríguez and Nivre, 2010, *inter alia*). These systems are, however, restricted to certain subsets of non-projective structures. In contrast, SwapStandard imposes no such restrictions and is able to parse unrestricted non-projective structures.

Dynamic oracles were first introduced by Goldberg and Nivre (2012) for the ArcEager system. They also proposed the standard way of exploiting dynamic oracles for training greedy parsers known as *training with exploration*. Here, the idea is that sometimes erroneous transitions are predicted during training. The dynamic oracle then comes into play by guiding the model towards the best possible tree, subject to the mistakes that have already been made. However, search-based parsers model the parsing problem as a structured prediction problem and are trained to predict optimal sequences of transitions for an entire sentence. Training with exploration is thus not applicable.

More recent work on dynamic oracles has primarily focused on developing dynamic oracles for other transition systems. Goldberg et al. (2014) present dynamic oracles for the ArcStandard system and the LR-spine parser by Sartorio et al. (2013). The only dynamic oracle for non-projective dependency trees was introduced by Gómez-Rodríguez et al. (2014) for a special case of Attardi's (2006) system. To date no dynamic oracle has been presented for transition systems that can handle unrestricted non-projective dependencies.

The underlying idea in our work is that there may be more than a single decomposition (i.e., transition sequence) that can recover the correct output (dependency tree). Rather than selecting a single unique such decomposition, the choice of decomposition can be thought of as **latent** and deferred to the machine learning algorithm. This approach has been shown to be successful for a number of tasks, including coreference resolution (Fernandes et al., 2012), semantic parsing (Zhou et al., 2013), and statistical machine translation (Yu et al., 2013), to name a few.

3 Transition System

We begin by describing our notation and the SwapStandard system. For simplicity we omit the inclusion of arc labels from this description, although for the experimental evaluation we implement a labeled version of this system. For a more formal description of the system, as well as proofs of soundness and completeness, we refer the reader to Nivre (2009).

The SwapStandard system operates on *configurations* $c = (\Sigma, B, A)$, where Σ denotes a stack of partially processed tokens, B denotes a buffer of remaining input tokens, and A is a set of arcs. We denote stack items by $s_i, i \geq 0$ where s_0 denotes the topmost item on the stack. Similarly, let $b_i, i \geq 0$ denote the items of the buffer, b_0 denoting the first element on the buffer. Finally, let $h \rightarrow d$ denote an arc from the head h to the dependent d.

The system begins in an *initial configuration* $c_0 = ([0], [1, 2, 3, ...], \emptyset)$, where the stack consists solely of the root token (numbered 0), all input tokens are on the buffer (numbered 1, 2, ...), and the arc set is the empty set. A configuration is *terminal* when the buffer is empty and the stack consists only of the root token 0.

The possible transitions of the system are

- Shift (`SH`) – removes b_0 from the buffer and pushes it onto the stack,
- LeftArc (`LA`) – introduces an arc $s_0 \rightarrow s_1$ and removes s_1 from the stack,
- RightArc (`RA`) – introduces an arc $s_1 \rightarrow s_0$ and removes s_0 from the stack,
- Swap (`SW`) – removes s_1 from the stack and places it as the first element on the buffer.

The `SW` transition reorders tokens from the input on the fly, enabling the system to recover non-projective trees. Informally, a dependency tree is

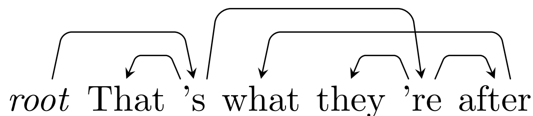

Figure 1: A non-projective dependency tree.

non-projective if it cannot be drawn without any crossing arcs. An example sentence with a non-projective tree is shown in Figure 1.

The total number of transitions required to parse a sentence of length n is bounded from below by $2n$ since every token needs to be shifted onto the stack once and attached to its head through an LA or RA once. Additionally, every SW moves a token back onto the buffer which subsequently needs to be shifted again, adding $2k$ transitions for k SW.

Eager oracle. Nivre (2009) presents a static oracle for the SwapStandard system. A high-level algorithmic description of this oracle is shown in Algorithm 1. The functions CANLA, CANRA, and CANSW define the requirements for the corresponding transitions. For LA the requirement is that s_0 is the head of s_1 and that s_1 has already collected all of its own dependents, and vice versa for RA.

To decide when SW can be applied, Nivre (2009) introduces the notion of **projective order** which is obtained through an inorder traversal of the dependency tree. The projective order is a total order over the tokens of a sentence. If the tokens are sorted accordingly, the tree becomes projective. The SW transition is allowed when s_0 precedes s_1 according to the projective order.

For a given sentence x, which is understood to include a representation of x's dependency tree and projective order, Algorithm 1 can be used to create a sequence of transitions that can derive the corresponding dependency tree. Specifically, iteratively call the oracle starting from x's initial configuration until the terminal configuration has been reached. This transition sequence is the oracle sequence for x.

In the example from Figure 1, the projective order is *That* $<$ *'s* $<$ *they* $<$ *'re* $<$ *what* $<$ *after*. The difference compared to the original word order is that *what* has been moved two tokens to the right. This means that SW is permissible when either of *they* or *'re* are s_0 and *what* is s_1. The oracle would apply SW when *what* is s_1 and *they* is s_0. It would then continue with two more SH followed by another SW, thereby obtaining the projective order. Since this oracle applies SW whenever the projective order admits it, we refer to it as EAGER.

Algorithm 1 Generic static oracle

```
Input: Configuration c, sentence x
1: if CanLA(c, x) then
2:     return LA
3: else if CanRA(c, x) then
4:     return RA
5: else if CanSW(c, x) then
6:     return SW
7: else
8:     return SH
```

Lazy oracle. For non-projective trees, the sequence given by EAGER is often not the shortest sequence, and shorter sequences that require fewer SW transitions to produce the same parse may be possible. Drawing upon this observation, Nivre et al. (2009) present an improved oracle that considerably reduces the number of swaps. Their oracle is based on the same basic structure as the one in Algorithm 1, but the semantics of CANSW are redefined. The oracle relies on a notion of **maximal projective components** (MPCs). In addition to requiring that s_0 and s_1 appear in the wrong order with respect to the projective order, this oracle also requires s_0 and s_1 not to be part of the same MPCs. MPCs are defined as the resulting subtrees obtained by running the oracle parser without SW until it hits a dead end. Nivre et al. (2009) show that this oracle substantially reduces the number of SW transitions, sometimes by up to 80%.

In the example from Figure 1 this oracle would not admit the first SW transition when *what* is s_1 and *they* is s_0. Instead, it would continue by shifting *'re* and then attaching *they* through an LA. As this oracle tries to postpone SW transitions when possible, we refer to it as LAZY.

4 Spurious Ambiguities

The static oracles presented above can produce a sequence of transitions to derive any dependency tree. By design, the algorithm selects among the possible transitions using a pre-defined order, i.e., the order of the tests in the if-clauses. This procedure yields a deterministic sequence of transitions for any dependency tree. This sequence is not necessarily the only correct one for a given dependency tree. In fact, most dependency trees seen in standard treebanks can be derived from more than one sequence of transitions. Different such

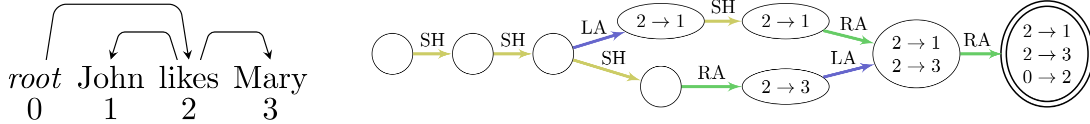

Figure 2: A dependency tree which exhibits the SH-LA ambiguity (left) and a lattice that encodes the two alternative transition sequences (right).

sequences always start and end with the same transitions, however, somewhere along the way a fork point occurs where more than one transition can be applied while still keeping the desired dependency tree reachable. We call these fork points *spurious ambiguities* as they all lead to the same tree. We have already seen one such ambiguity above, comparing EAGER and LAZY, namely the SH-SW ambiguity. Below we review the remaining spurious ambiguities of the SwapStandard system and give examples of each kind.

SH-LA ambiguity. While the canonical way of constructing the dependency tree is to attach left dependents as early as possible, these decisions can sometimes be delayed. Consider the example sentence on the left in Figure 2. Here, the left dependent of *likes* need not be attached before the right. The example thus has two possible transition sequences that create the given dependency tree. These sequences can be illustrated in the form of a lattice, as depicted on the right in Figure 2. Nodes in the lattice correspond to parser configurations and arcs between them to transitions (color-coded for different transitions). The initial configuration is to the left, and the terminal configuration is on the right (indicated by a double circle). The text in the nodes show the arcs that have been constructed thus far.[1] The SH-LA ambiguity gives rise to the fork point where there are two parallel paths, corresponding to early and late attachments of the left dependent. This ambiguity is not specific to SwapStandard, but also occurs in the projective ArcStandard system. In the projective case, this type of ambiguity always arises when the parser can make an LA attachment but b_0 is dominated by s_0.

SH-RA ambiguity. While the SH-RA ambiguity is not possible in the plain ArcStandard system the introduction of SW enables this ambiguity. In the ArcStandard system, applying an SH when an RA

[1] While the nodes only display the arc set so far, the merge points in the lattice truly correspond to equivalent states where the stack, buffer, and arc set are all identical.

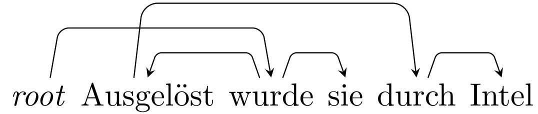

Figure 3: A non-projective dependency tree which exhibits the SH-RA ambiguity.

is possible results in "burying" tokens on the stack such that they are irretrievable. Since the SW transition moves tokens out of the stack and back onto the buffer, it is sometimes possible to recover these buried tokens and thus do the RA at a later point.

Consider the German sentence in Figure 3. The static oracle would parse this sentence by first applying three SH followed by an RA, attaching *sie* to *wurde*. However, the projective order of this sentence is *Ausgelöst* < *durch* < *Intel* < *wurde* < *sie*. The system is thus able to delay the RA transition, do another SH and then swap *wurde* and *sie* past *durch* and handle this attachment later. The lattice in Figure 4 shows all ambiguities for this sentence. The first SH-RA ambiguity is highlighted.

In comparison to the SH-LA ambiguity seen earlier, the SH-RA ambiguity always relies on additional SW transitions and thus leads to longer transition sequences. Note that the same logic also holds for LA – sometimes both the head and dependent of an LA transition can be swapped out, creating a second kind of SH-LA ambiguity in addition to the one seen already.

4.1 Non-deterministic oracles

Recall that the main hypothesis of our work is that a search-based parser can be further improved by using a non-deterministic oracle. So far we have seen three types of spurious ambiguities. It is easy to convince oneself that no other ambiguities are possible – by sheer enumeration of the possible pairs of transitions, any other ambiguities would involve one of the pairs LA-RA, LA-SW, or RA-SW. An LA-RA ambiguity would only be possible if s_0 is the head of s_1 and s_1 is the head of s_0 which implies that the tree has a cycle. The LA-SW and

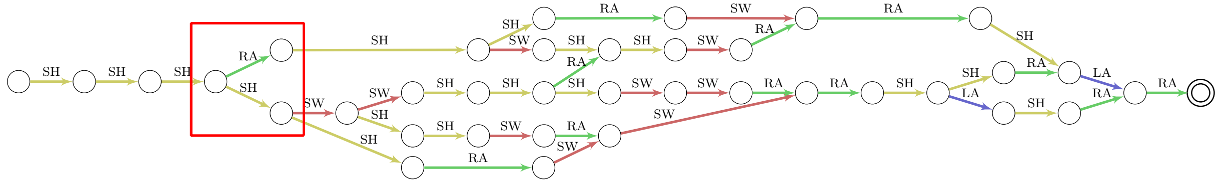

Figure 4: A lattice which encodes all possible transition sequences to parse the sentence from Figure 3. The initial SH-RA ambiguity is highlighted in the red box.

Algorithm 2 Can shift

Input: Configuration c, sentence x
1: **if** $|c.B| = 0$ **then** ▷ Not possible if buffer is empty
2: **return** false
3: $c = \text{DoSH}(c)$ ▷ Initial shift
4: **while** $\neg\text{TERMINAL}(c)$ **do**
5: **if** $\text{CANLA}(c, x)$ **then**
6: $c = \text{DoLA}(c)$
7: **else if** $\text{CANRA}(c, x)$ **then**
8: $c = \text{DoRA}(c)$
9: **else if** $\text{CANSW}_{\text{Eager}}(c, x)$ **then**
10: $c = \text{DoSW}(c)$
11: **else if** $|c.B| > 0$ **then** ▷ SH if buffer is not empty
12: $c = \text{DoSH}(c)$
13: **else**
14: **return** false ▷ Hit dead end
15: **return** true

RA-SW ambiguities would swap a dependent past its head, or a head past its dependent. This would, however, violate the projective order and is therefore also not possible.

The key question that needs to be resolved in order to construct a non-deterministic oracle is when an SH transition can be applied. The EAGER oracle defines when the other three transitions can be applied, but treats SH as a fallback when no other transitions are possible. So when is SH applicable? One way to find out is to try an SH and see if there is any way to recover the full parse. This procedure is described in Algorithm 2. The algorithm applies an initial shift (line 3) and uses the EAGER oracle from then on (lines 4 to 14). If the parser can recover the correct parse, then SH is permissible. If not, the parser will eventually end up in a dead end where the buffer is empty, the stack contains several items, but no other transitions are applicable (line 14).[2]

The reason this algorithm works is that after applying the initial SH, it prefers all the other transitions over additional SH transitions. The other transitions are all taking clear steps towards avoiding dead ends, either by introducing arcs (removing tokens from the system) or by applying swaps (permuting the words in the direction of the projective order by moving tokens back from the stack onto the buffer).

The procedure outlined in Algorithm 2 allows us to construct a non-deterministic oracle for the SwapStandard system. Specifically, whenever either of LA, RA, or SW are permissible, the oracle also checks if SH is possible. If neither of LA, RA, or SW are permissible, SH is returned. This oracle allows for all possible ambiguities and we refer to it as ND-ALL.

It could be argued that a parser could profit from having a more eager treatment of arc attachments and that the SH-LA and SH-RA ambiguities should be avoided. As for the SH-SW ambiguity, we have seen that there is a continuum of how eagerly SW transitions should be applied, ranging from EAGER to LAZY (and beyond, as we will see shortly). The static oracles apply SW according to the predefined rules that are primarily grounded in tree structural characteristics. These rules may not be the most motivated from a linguistic perspective, and there could be a more systematic treatment of swaps lying somewhere in between that is easier to learn. In order to investigate whether the parser is able to learn better such patterns latently, we also construct an oracle that only permits the SH-SW ambiguity but no others. We will refer to this oracle as ND-SW.

4.2 Minimally swapping oracle

While the LAZY oracle considerably reduces the number of SW transitions, this oracle still does not always yield the minimal number of swaps for a given dependency tree. Given the possibility to tell when an SH is possible (Algorithm 2), we can construct lattices as those shown above for any dependency tree. By searching this lattice for the shortest path from the initial state to the terminal

[2] The worst case runtime of this algorithm is $O(n^2)$, although it is enough to halt the search when the stack has been reduced to only two tokens (one being root), since from that point on the gold tree has to be recoverable. In practice we observed that applying Algorithm 2 during training had negligible effect on overall training time.

state, the shortest possible transition sequence can be found. As this oracle minimizes the number of SW transitions, we refer to it as MINIMAL.[3]

This procedure cannot be formalized as concisely as Algorithm 1 and relies on searching the corresponding lattice. But it should be noted that when a static oracle is used for training, the transition sequence for each sentence only needs to be computed once before training.

The lattices can grow extremely large, to the point where the lattice of a single sentence cannot be kept in main memory.[4] A depth-first search that keeps a stack of fork points in memory circumvents this problem. After reaching the terminal state the first time, only the path chosen and the number of transitions need to be remembered. Any further paths that have not reached the terminal state after as many transitions as the currently seen shortest path can be terminated immediately. While this oracle is clearly slower than the other static ones, we found that the extra overhead is negligible in comparison to overall training time.

5 Training

We train the parser with a variant of the structured perceptron (Collins, 2002) and use a beam size of 20. We follow Bohnet et al. (2013) and use the the Passive-Aggressive algorithm (Crammer et al., 2006). We deviate slightly from the previous work and use the Max-Violation framework (Huang et al., 2012) rather than early update (Collins and Roark, 2004), as we found that it required fewer iterations and yielded slightly higher scores, both for static and non-deterministic oracles. Following standard practice, we also apply parameter averaging (Collins, 2002).

When training with a static oracle the correct configuration to update against is well-defined, but with a non-deterministic oracle there may be more than one correct configuration and it is unclear against which to update. Yu et al. (2013) suggest to compare with the highest scoring correct configuration at every step but in initial experiments we found that this performed rather poorly. Instead, we apply beam search in a constrained setting to arrive at a single best correct sequence using the current parameters. This sequence is the *latent* gold sequence and is recomputed for every sentence during every iteration.

When we train a greedy parser we fall back to the standard perceptron algorithm using a non-deterministic oracle (Goldberg and Nivre, 2013; Goldberg et al., 2014). Since this model is not globally trained, only the choice of the next transition is latent but the basic principle is the same.

The feature model we use is primarily based on that of Zhang and Nivre (2011) with the obvious adaptations to the SwapStandard setting. Additional features are taken from other recent work on parsers using the SwapStandard system (Bohnet and Nivre, 2012; Bohnet et al., 2013). Following the line of work presented by Bohnet et al. we also replace the feature mapping function by a hash function which enables the use of negative features and yields a considerable speed improvement (Bohnet, 2010).[5]

6 Experiments

In total we experiment with five different oracles. The three static ones, EAGER, LAZY, and MINIMAL, all use a single unique transition sequence for every sentence in the training data. The two non-deterministic oracles, ND-SW and ND-ALL, create latent transition sequences on the fly relying on the current parameters and may change across training iterations. Our main hypothesis is that the latent sequences created by the non-deterministic oracles are easier to learn and generalize better to unseen data, leading to increased accuracy.

Data sets. We evaluate the oracles on ten treebanks. Specifically, we use the nine treebanks from the SPMRL 2014 Shared Task (Seddah et al., 2014), comprising Arabic, Basque, French, German, Hebrew, Hungarian, Korean, Polish, and Swedish. For these treebanks we use the train/dev/test splits provided by the Shared Task organizers. Additionally, we use the English Penn Treebank (Marcus et al., 1993) converted to Stanford dependencies (de Marneffe et al., 2006) with the standard split, sections 2-21 for training, section 24 for development, and section 23 for test.

A breakdown of the characteristics of the training sets of each treebank is shown in Table 1. The

[3] It should be noted that in certain cases the number of SW transitions can be reduced even further by swapping tokens that are already in the projective order. The MINIMAL oracle ensures that the number of SW transitions is minimal while still respecting the projective order.

[4] Even with 256gb of main memory we were unable to keep some of the lattices of the training sets in memory despite an efficient implementation.

[5] For the sake of reproducibility we make our implementation available on the first author's website.

	ar	de	en	eu	fr	he	hu	ko	pl	sv
# Sent.	15,762	40,472	39,832	7,577	14,759	5,000	8,146	23,010	6,578	5,000
% Proj. Sent.	97.32%	67.23%	99.90%	94.71%	99.97%	99.82%	87.75%	100%	99.54%	93.62%
# EAGER Swaps	6,481	155,041	146	984	6	12	3,654	0	91	2,062
% Swap Red. (LAZY)	80.59%	75.09%	71.92%	53.46%	16.67%	8.33%	51.07%	-	59.34%	75.90%
% Swap Red. (MINIMAL)	80.79%	83.88%	-	-	-	-	54.24%	-	-	77.79%
% Sent. w/ Unique tr. seq.	9.94%	7.81%	1.31%	1.06%	2.66%	2.82%	10.25%	0.27%	10.57%	7.28%

Table 1: Data set statistics (training sets).

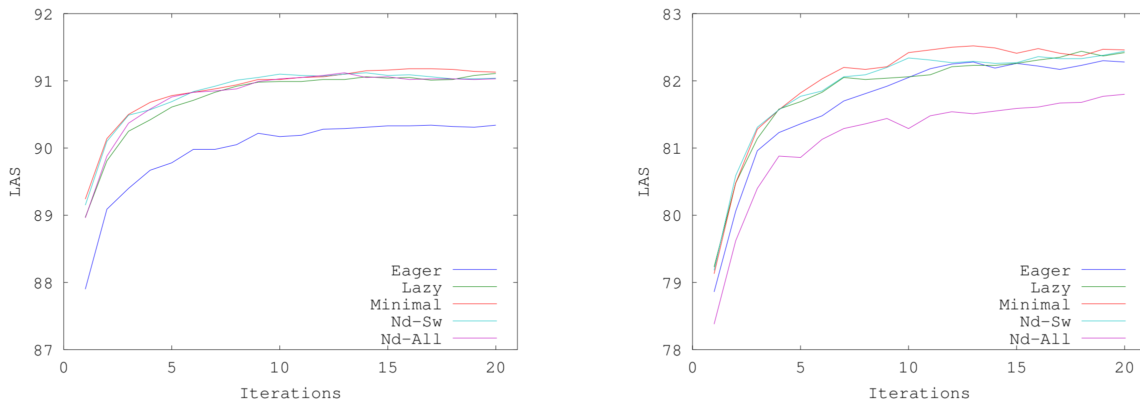

Figure 5: Learning curves of the different oracles for German (left) and Hungarian (right).

table shows the total number of sentences and the percentage of projective sentences. It also shows the total number of swap transitions required by EAGER, and the reduction of swaps of LAZY and MINIMAL relative to EAGER. For instance, in the Arabic treebank 97.32% of the sentences are projective and the LAZY and MINIMAL reduce the number of swaps by 80.59% and 80.79%, respectively. For about half the treebanks LAZY is already minimal and we exclude MINIMAL.

Korean is the only strictly projective treebank, although some of the treebanks have very few non-projective arcs in their training sets, particularly Hebrew and French. This means that the number of `SH-SW` ambiguities considered by the non-deterministic oracles during training is extremely small. The ND-SW oracle thus exhibits a very tiny amount of spurious ambiguity in these cases. Nevertheless, ND-ALL will still consider the `SH-LA` ambiguity. The last row of Table 1 shows the percentage of sentences that exhibit no spurious ambiguity under the ND-ALL oracle. This fraction ranges between almost 0% and up to about 10%, which means that there are indeed plenty of spurious ambiguities in the training data.

Preprocessing. We adopt a realistic evaluation setting and use predicted part-of-speech tags and morphological features. Specifically, we use MarMoT (Mueller et al., 2013), a state-of-the-art CRF tagger that jointly predicts part-of-speech tags and morphology. We train the parsers on 10-fold jackknifed training data. For the development and test sets the tagger is trained on the full training set.

Evaluation. We evaluate the parsers using labeled attachment score (LAS), i.e., the percentage of arcs that have the correct heads and labels. We omit the unlabeled version of this metric as we observed that it is closely correlated with LAS. We test for significance using the Wilcoxon signed rank test and mark significance at the $p < 0.05$ and $p < 0.01$ levels with † and ‡, respectively.

Training iterations. Since the parsers trained using the non-deterministic oracles rely on a latent sequence, they might require more training iterations before reaching good performance. Moreover, during initial experiments on the development data we saw that the learning curves are not monotonically increasing. To test our main hypothesis – that beam-search parsers can profit from training with a non-deterministic oracle – we tune the number of training iterations on the development sets for each oracle and treebank.

Figure 5 shows the learning curves of the beam search parser on German and Hungarian. These two plots are chosen since they paint a rather divergent picture, where EAGER is clearly underperforming for German, and ND-ALL is considerably worse than other oracles for Hungarian. For most of the other treebanks, however, the learning curves are surprisingly similar for all oracles.

	ar	de	en	eu	fr	he	hu	ko	pl	sv
ND-SW	85.92	**91.12**	**89.08**	80.53	83.49	77.89	**82.44**	-	82.50	**74.75**
ND-ALL	**86.10**	**91.12**	88.80	**80.88**	**83.66**	**78.00**	81.80	**85.18**	**83.98**	74.60
EAGER	85.88	90.34	**88.96**	80.54	83.49	77.85	82.30	**85.30**	82.74	75.01
LAZY	85.93	91.11	88.94	**80.87**	**83.65**	**77.99**	82.44	-	**82.99**	75.25
MINIMAL	**85.96**	**91.18**	-	-	-	-	**82.52**	-	-	**75.41**

Table 2: Beam search results on dev sets. The best non-deterministic and static oracles are bold.

	ar	de	en-sd	eu	fr	he	hu	ko	pl	sv
Static	85.05	87.53	90.35	79.97	83.10	78.65	83.60	85.03	82.08	79.05
Non-det.	+0.06	-0.23	+0.13	+0.55	-0.11	-0.39	+0.08	+0.09	+1.26‡	-0.07

Table 3: Test set result with beam search comparing the best static and non-deterministic oracles.

6.1 Results

Table 2 displays the LAS on the development sets for each oracle after tuning. When LAZY is already minimal, we omit MINIMAL. Since Korean is projective, we only compare ND-ALL and EAGER, which thus reduces to comparing a static and a non-deterministic oracle for ArcStandard.

The differences between the oracles are rather small. The most interesting differences occur for German, where the EAGER oracle is clearly behind, and Polish, where ND-ALL is considerably ahead of all the other oracles. Polish is also the only case where one of the non-deterministic oracles appears to be clearly ahead of the static ones.

Among the static oracles the oracle that requires the least amount of SW transitions generally performs best. The only exception is English, where EAGER is marginally ahead of LAZY. While the English treebank has relatively few non-projective sentences, the case is even more extreme for Hebrew and French. For these treebanks the difference between EAGER and LAZY amounts to a single swap, yet the difference in LAS is greater than 0.1%. This tiny difference in transition sequences in the training data appears to have a butterfly effect during the online learning, such that a single different update changes the outcome of the resulting weight vector to this effect.

For the non-deterministic oracles the picture is more mixed. Which oracle is better appears to be rather treebank specific, although for the most part the differences are not that big.

Finally we compare the best static with the best non-deterministic oracle on the test sets of each language. The results are shown in Table 3. In about half the cases the non-deterministic oracle does slightly better than the static one, but in the other half it is the other way around. The only significant difference is the improvement of the non-deterministic oracle for Polish. All in all, however, we conclude that static oracles generally perform as well as non-deterministic oracles.

6.2 What about greedy?

Since the non-deterministic oracles do not seem to be that helpful for the beam search parser, we wonder if the effect is the same in the greedy setting. This case has previously only been studied for projective parsers (Goldberg and Nivre, 2012; Goldberg et al., 2014). Table 4 shows the results of the greedy parser on the development sets after tuning. The greedy parser exhibits a clearer pattern compared to the beam search parser. For the non-deterministic oracles, ND-ALL is typically the best. For the static oracles the trend is that fewer swaps are better.

The final evaluation on the test sets of the greedy parser is shown in Table 5. In four cases the non-deterministic oracle is significantly better than the best static one, and overall the non-deterministic oracle is never harmful.

Comparing the results between the greedy and beam search parsers, it is clear that the greedy parser generally is behind by one or two points absolute. A peculiar exception is Korean, where the differences between the two parsers are remarkably small, yet in favor of the beam search parser.

6.3 Discussion

So what do the latent transition sequences look like? Figure 6 shows two plots of the average number of SW transitions per sentence for German and Hungarian as a function of the number of training iterations. The static oracles render straight lines since the transition sequences do not change between iterations, while the non-deterministic oracles do. Also here these two treebanks exhibit different extremes. In the case of German, the EAGER oracle is clearly applying SW much more

	ar	de	en	eu	fr	he	hu	ko	pl	sv
ND-SW	83.84	88.44	86.83	79.34	81.57	75.75	**79.17**	-	80.31	**72.74**
ND-ALL	**84.12**	**88.76**	**87.57**	**79.59**	**81.99**	**76.18**	78.87	**85.06**	**80.47**	72.72
EAGER	83.68	87.45	86.55	79.07	81.52	75.62	79.04	**84.92**	**80.18**	72.53
LAZY	83.74	88.45	**86.70**	**79.40**	**81.57**	**75.75**	**79.18**	-	79.60	**73.29**
MINIMAL	**83.76**	**88.81**	-	-	-	-	79.15	-	-	73.08

Table 4: Greedy results on dev sets. The best non-deterministic and static oracles are bold.

	ar	de	en	eu	fr	he	hu	ko	pl	sv
Static	82.99	84.22	87.85	78.58	81.12	75.27	81.45	84.52	79.10	75.89
Non-det.	+0.04	+0.03	+0.60‡	+0.24	+0.40‡	+0.70†	+0.22	+0.30	+1.33‡	+0.39

Table 5: Test set result with greedy search comparing the best static and non-deterministic oracles.

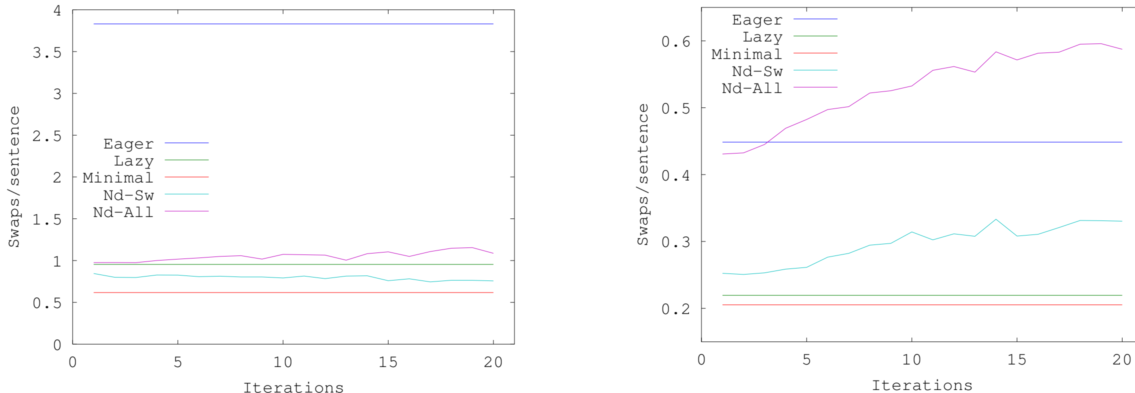

Figure 6: Average number of swaps per sentence during training for German (left) and Hungarian (right).

than any other oracle. The non-deterministic oracles tend to stay very close to the minimal number of SW transitions. For Hungarian the picture is dramatically different. The ND-ALL oracle has a tendency to overswap and gradually applies more and more SW transitions. These results bear a striking resemblance to those shown in the learning curves from Figure 5, where EAGER and ND-ALL are bad for German and Hungarian, respectively. For most of the other treebanks the corresponding curves are much closer. Indeed, as the other treebanks exhibit considerably less non-projectivity, the amount of spurious ambiguity and choice of swaps is much more constrained.

But why do the non-deterministic oracles seem to be beneficial for the greedy parser but not for the beam search parser? One reason might be that the non-deterministic oracle provides greater diversity in the training data. This is the same effect that dynamic oracles achieve with training by exploration, although it is less pronounced when only using a non-deterministic oracle. The beam search parser, on the other hand, is already exposed to many mistakes during training because of the global learning. Since the beam search parser actually does explore multiple possible transition sequences, it is probably also more lenient towards only seeing a single (static) sequence of transitions for every training instance.

7 Conclusion

The SwapStandard transition system for dependency parsing has proven very useful, especially for parsing languages with a high degree of non-projectivity, but until now it has not been known how to define non-deterministic oracles for this system. By mapping out the spurious ambiguities of the system, we have managed to solve this problem as well as the open problem of finding the minimal number of swaps using a static oracle. This has enabled us, for the first time, to evaluate the utility of non-deterministic oracles when training non-projective dependency parsers using beam search as well as greedy search. In the beam search case, the results indicate that there is no real benefit non-deterministic oracles, presumably because beam search compensates for the non-determinism. In the greedy case, we have extended previous results to the non-projective domain, showing that non-deterministic oracles are at least as good as, and sometimes significantly better than, static ones.

Acknowledgments

We are grateful to the anonymous reviewers and Agnieszka Faleńska for comments on earlier versions of this paper. We also thank Wolfgang Seeker for many valuable discussions. The first author is funded by the Deutsche Forschungsgemeinschaft (DFG) via SFB 732, project D8.

References

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 166–170, New York City, June. Association for Computational Linguistics.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465, Jeju Island, Korea, July. Association for Computational Linguistics.

Bernd Bohnet, Joakim Nivre, Igor Boguslavsky, Richárd Farkas, Filip Ginter, and Jan Hajič. 2013. Joint morphological and syntactic analysis for richly inflected languages. *Transactions of the Association for Computational Linguistics*, 1:415–428.

Bernd Bohnet. 2010. Top accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 89–97, Beijing, China, August. Coling 2010 Organizing Committee.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 111–118, Barcelona, Spain, July.

Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8. Association for Computational Linguistics, July.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive–aggressive algorithms. *Journal of Machine Learning Reseach*, 7:551–585, March.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, pages 449–454.

Eraldo Fernandes, Cícero dos Santos, and Ruy Milidiú. 2012. Latent structure perceptron with feature induction for unrestricted coreference resolution. In *Joint Conference on EMNLP and CoNLL - Shared Task*, pages 41–48, Jeju Island, Korea, July. Association for Computational Linguistics.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India, December. The COLING 2012 Organizing Committee.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 1:403–414.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2:119–130.

Carlos Gómez-Rodríguez and Joakim Nivre. 2010. A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1492–1501, Uppsala, Sweden, July. Association for Computational Linguistics.

Carlos Gómez-Rodríguez, Francesco Sartorio, and Giorgio Satta. 2014. A polynomial-time dynamic oracle for non-projective dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 917–927, Doha, Qatar, October. Association for Computational Linguistics.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151, Montréal, Canada, June. Association for Computational Linguistics.

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

Thomas Mueller, Helmut Schmid, and Hinrich Schütze. 2013. Efficient higher-order CRFs for morphological tagging. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 322–332, Seattle, Washington, USA, October. Association for Computational Linguistics.

Joakim Nivre, Marco Kuhlmann, and Johan Hall. 2009. An improved oracle for dependency parsing with online reordering. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, pages 73–76, Paris, France, October. Association for Computational Linguistics.

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

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

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

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. 2013. A transition-based dependency parser using a dynamic parsing strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 135–144, Sofia, Bulgaria, August. Association for Computational Linguistics.

Djamé Seddah, Sandra Kübler, and Reut Tsarfaty. 2014. Introducing the spmrl 2014 shared task on parsing morphologically-rich languages. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, pages 103–109, Dublin, Ireland, August. Dublin City University.

Heng Yu, Liang Huang, Haitao Mi, and Kai Zhao. 2013. Max-violation perceptron and forced decoding for scalable MT training. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1112–1123, Seattle, Washington, USA, October. Association for Computational Linguistics.

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 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571, Honolulu, Hawaii, October. Association for Computational Linguistics.

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, Portland, Oregon, USA, June. Association for Computational Linguistics.

Junsheng Zhou, Juhong Xu, and Weiguang Qu. 2013. Efficient latent structural perceptron with hybrid trees for semantic parsing. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 2246–2252. AAAI Press.