Dynamic Oracles for Top-Down and In-Order Shift-Reduce Constituent Parsing

Daniel Fernández-González and **Carlos Gómez-Rodríguez**
Universidade da Coruña
FASTPARSE Lab, LyS Research Group, Departamento de Computación
Campus de Elviña, s/n, 15071 A Coruña, Spain
d.fgonzalez@udc.es, carlos.gomez@udc.es

Abstract

We introduce novel dynamic oracles for training two of the most accurate known shift-reduce algorithms for constituent parsing: the top-down and in-order transition-based parsers. In both cases, the dynamic oracles manage to notably increase their accuracy, in comparison to that obtained by performing classic static training. In addition, by improving the performance of the state-of-the-art in-order shift-reduce parser, we achieve the best accuracy to date (92.0 F1) obtained by a fully-supervised single-model greedy shift-reduce constituent parser on the WSJ benchmark.

1 Introduction

The shift-reduce transition-based framework was initially introduced, and successfully adapted from the dependency formalism, into constituent parsing by Sagae and Lavie (2005), significantly increasing phrase-structure parsing performance.

A shift-reduce algorithm uses a sequence of transitions to modify the content of two main data structures (a buffer and a stack) and create partial phrase-structure trees (or constituents) in the stack to finally produce a complete syntactic analysis for an input sentence, running in linear time. Initially, Sagae and Lavie (2005) suggested that those partial phrase-structure trees be built in a *bottom-up* manner: two adjacent nodes already in the stack are combined under a non-terminal to become a new constituent. This strategy was followed by many researchers (Zhang and Clark, 2009; Zhu et al., 2013; Watanabe and Sumita, 2015; Mi and Huang, 2015; Crabbé, 2015; Cross and Huang, 2016b; Coavoux and Crabbé, 2016; Fernández-González and Gómez-Rodríguez, 2018) who managed to improve the accuracy and speed of the original Sagae and Lavie's bottom-up parser. With this, shift-reduce algorithms have become competitive, and are the fastest alternative to perform phrase-structure parsing to date.

Some of these attempts (Cross and Huang, 2016b; Coavoux and Crabbé, 2016; Fernández-González and Gómez-Rodríguez, 2018) introduced *dynamic oracles* (Goldberg and Nivre, 2012), originally designed for transition-based dependency algorithms, to bottom-up constituent parsing. They propose to use these dynamic oracles to train shift-reduce parsers instead of a traditional *static oracle*. The latter follows the standard procedure that uses a gold sequence of transitions to train a model for parsing new sentences at test time. A shift-reduce parser trained with this approach tends to be prone to suffer from error propagation (i.e. errors made in previous states are propagated to subsequent states, causing further mistakes in the transition sequence). Dynamic oracles (Goldberg and Nivre, 2012) were developed to minimize the effect of error propagation by training parsers under closer conditions to those found at test time, where mistakes are inevitably made. They are designed to guide the parser through any state it might reach during learning time. This makes it possible to introduce error exploration to force the parser to go through non-optimal states, teaching it how to recover from mistakes and lose the minimum number of gold constituents.

Alternatively, some researchers decided to follow a different direction and explore non-bottom-up strategies for producing phrase-structure syntactic analysis.

On the one hand, (Dyer et al., 2016; Kuncoro et al., 2017) proposed a *top-down* transition-based algorithm, which creates a phrase structure tree in the stack by first choosing the non-terminal on the top of the tree, and then considering which should be its child nodes. In contrast to the bottom-up approach, this top-down strategy adds a lookahead

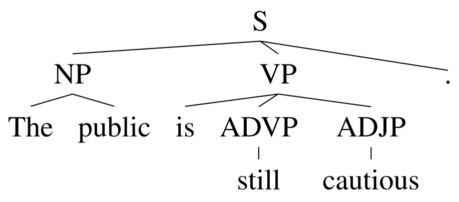

Figure 1: Simplified constituent tree, taken from English WSJ §22.

guidance to the parsing process, while it loses rich local features from partially-built trees.

On the other hand, Liu and Zhang (2017a) recently developed a novel strategy that finds a compromise between the strengths of top-down and bottom-up approaches, resulting in state-of-the-art accuracy. Concretely, this parser builds the tree following an *in-order* traversal: instead of starting the tree from the top, it chooses the non-terminal of the resulting subtree after having the first child node in the stack. In that way each partial constituent tree is created in a bottom-up manner, but the non-terminal node is not chosen when all child nodes are in the stack (as a purely bottom-up parser does), but after the first child is considered.

Liu and Zhang (2017a) report that the top-down approach is on par with the bottom-up strategy in terms of accuracy and the in-order parser yields the best accuracy to date on the WSJ. However, despite being two adequate alternatives to the traditional bottom-up strategy, no further work has been undertaken to improve their performance.[1]

We propose what, to our knowledge, are the first optimal dynamic oracles for both the top-down and in-order shift-reduce parsers, allowing us to train these algorithms with exploration. The resulting parsers outperform the existing versions trained with static oracles on the WSJ Penn Treebank (Marcus et al., 1993) and Chinese Treebank (CTB) benchmarks (Xue et al., 2005). The version of the in-order parser trained with our dynamic oracle achieves the highest accuracy obtained so far by a single fully-supervised greedy shift-reduce system on the WSJ.

2 Preliminaries

The original transition system of Sagae and Lavie (2005) parses a sentence from left to right by reading (moving) words from a *buffer* to a *stack*, where partial subtrees are built. This process is performed by a sequence of Shift (for reading) and Reduce (for building) transitions that will lead the parser through different states or parser configurations until a terminal one is reached. While in the bottom-up strategy the Reduce transition is in charge of labeling the partial subtree with a non-terminal at the same time the tree is built, Dyer et al. (2016) and Liu and Zhang (2017a) introduce a novel transition to choose the non-terminal on top, leaving the Reduce transition just to create the subtree under the previously decided non-terminal. We will now explain more in detail both the top-down and the in-order transition systems.

In both transition systems, parser configurations have the form $c = \langle \Sigma, i, f, \gamma, \alpha \rangle$, where Σ is a stack of constituents, i is the position of the leftmost unprocessed word in the buffer (which is the next to be pushed onto the stack), f is a boolean variable used by the in-order transition system to mark if a configuration is terminal or not and with no value in top-down parser configurations, γ is the set of constituents that have already been built, and α is the set of non-terminal nodes that are currently in the stack.

Each constituent is represented as a tuple (X, l, r), where X is a non-terminal and l and r are integers defining its span. Constituents are composed of one or several words or constituents, and just one non-terminal node on top. Each word w_i is represented as $(w, i, i+1)$. To define our oracles, we will need to represent each non-terminal node of the tree as (X, j), where j has the value of i when X is included in the stack and is used to keep them in order.[2]

For instance, the phrase-structure tree in Figure 1 can be decomposed as the following set of gold constituents: {(S, 0, 6), (NP, 0, 2), (VP, 2, 5), (ADVP, 3, 4), (ADJP, 4, 5)}. In addition, the ordered set of gold non-terminal nodes added to the stack while following a top-down strategy will be {(S, 0), (NP, 0), (VP, 2), (ADVP, 3), (ADJP, 4)} and, according to an in-order approach, {(NP, 1), (S, 2), (VP, 3), (ADVP, 4), (ADJP, 5)}. It is worth mentioning that the index of non-terminal nodes in the top-down method is the same as the leftmost span index of the constituent that it will produce. However, this does not hold in the in-order approach, as the leftmost child is fully processed before the node is added to the stack, so the index

[1]In parallel to this work, Fried and Klein (2018) present a non-optimal dynamic oracle for training the top-down parser.

[2]When two or more non-terminals share their labels within the tree, we use a secondary index to make them unique.

for the node will point to the leftmost span index of the second leftmost child.

Note that the information about the span of a constituent, the set of predicted constituents γ and the set α of predicted non-terminal nodes in the stack is not used by the original top-down and in-order parsers. However, we need to include it in parser configurations at learning time to allow an efficient implementation of the proposed dynamic oracles.

Given an input string $w_0 \cdots w_{n-1}$, the in-order parsing process starts at the initial configuration $c_s(w_0 \ldots w_{n-1}) = \langle [\,], 0, \mathit{false}, \{\}, \{\}\rangle$ and, after applying a sequence of transitions, it ends in a terminal configuration $\langle (S, 0, n), n, \mathit{true}, \gamma, \alpha\rangle$, where n is the number of words in the input sentence. The top-down transition system shares the same form for the initial and terminal configurations, except for the fact that variable f has no value in both cases.

Figure 2 shows the available transitions in the top-down algorithm. In particular, the Shift transition moves the first (leftmost) word in the buffer to the stack; the Non-Terminal-X transition pushes onto the stack the non-terminal node X that should be on top of a coming constituent, and the Reduce transition pops the topmost stack nodes until the first non-terminal node appears (which is also popped) and combines them into a constituent with this non-terminal node as their parent, pushing this new constituent into the stack. Note that every reduction action will add a new constituent to γ and remove a non-terminal node from α, and every Non-Terminal transition will include a new non-terminal node in α. Figure 3 shows the top-down transition sequence that produces the phrase-structure tree in Figure 1.

In Figure 4 we describe the available transitions in the in-order algorithm. The Shift, Non-Terminal-X and Reduce transitions have the same behavior as defined for the top-down transition system, except that the Reduce transition not only pops stack nodes until finding a non-terminal node (also removed from the stack), but also the node below this non-terminal node, and combines them into a constituent spanning all the popped nodes with the non-terminal node on top. And, finally, a Finish transition is also available to end the parsing process. Figure 5 shows the in-order transition sequence that outputs the constituent tree in Figure 1.

The standard procedure to train a greedy shift-reduce parser consists of training a classifier to approximate an *oracle*, which chooses optimal transitions with respect to gold parse trees. This classifier will greedily choose which transition sequence the parser should apply at test time.

Depending on the strategy used for training the parser, oracles can be static or dynamic. A static oracle trains the parser only on gold transition sequences, while a dynamic one can guide the parser through any possible transition path, allowing the exploration of non-optimal sequences.

3 Dynamic Oracles

Previous work such as (Cross and Huang, 2016b; Coavoux and Crabbé, 2016; Fernández-González and Gómez-Rodríguez, 2018) has introduced and successfully applied dynamic oracles for bottom-up phrase-structure parsing. We present dynamic oracles for training the top-down and in-order transition-based constituent parsers.

Goldberg and Nivre (2012) show that implementing a dynamic oracle reduces to defining a *loss function* on configurations to measure the distance from the best tree they can produce to the gold parse. This allows us to compute which transitions will lead the parser to configurations where the minimum number of mistakes are made.

3.1 Loss function

According to Fernández-González and Gómez-Rodríguez (2018), we can define a loss function in constituent parsing as follows: given a parser configuration c and a gold tree t_G, a loss function $\ell(c)$ is implemented as the minimum Hamming loss between t and t_G, ($\mathcal{L}(t, t_G)$), where t is the already-built tree of a configuration c' reachable from c (written as $c \rightsquigarrow t$). This Hamming loss is computed as the size of the symmetric difference between the set of constituents γ and γ_G in the trees t and t_G, respectively. Therefore, the loss function is defined as:

$$\ell(c) = \min_{\gamma | c \rightsquigarrow \gamma} \mathcal{L}(\gamma, \gamma_G) = |\gamma_G \setminus \gamma| + |\gamma \setminus \gamma_G|$$

and, according to the authors, it can be efficiently computed for a non-binary bottom-up transition system by counting the individually unreachable arcs from configuration c ($|\mathcal{U}(c, \gamma_G)|$) plus the erroneous constituents created so far ($|\gamma_c \setminus \gamma_G|$):

$$\ell(c) = \min_{\gamma | c \rightsquigarrow \gamma} \mathcal{L}(\gamma, \gamma_G) = |\mathcal{U}(c, \gamma_G)| + |\gamma_c \setminus \gamma_G|$$

Shift:	$\langle \Sigma, i, /, \gamma, \alpha \rangle \Rightarrow \langle \Sigma \vert (w_i, i, i+1), i+1, /, \gamma \cup \{(w_i, i, i+1)\}, \alpha \rangle$
Non-Terminal-X:	$\langle \Sigma, i, /, \gamma, \alpha \rangle \Rightarrow \langle \Sigma \vert (X, i), i, /, \gamma, \alpha \cup \{(X, i)\} \rangle$
Reduce:	$\langle \Sigma \vert (X, j) \vert (Y_1, m_0, m_1) \vert ... \vert (Y_k, m_{k-1}, m_k), i, /, \gamma, \alpha \rangle \Rightarrow \langle \Sigma \vert (X, j, m_k), i, /, \gamma \cup \{(X, j, m_k)\}, \alpha \setminus \{(X, j)\} \rangle$

Figure 2: Transitions of a top-down constituent parser.

Transition	Σ	Buffer
	[ ]	[The, ...]
NT$_S$	[S]	[The, ...]
NT$_{NP}$	[S, NP]	[The, ...]
SH	[S, NP, The]	[public, ...]
SH	[S, NP, The, public]	[is, ...]
RE	[S, **NP**]	[is, ...]
NT$_{VP}$	[S, **NP**, VP]	[is, ...]
SH	[S, **NP**, VP, is]	[still, ...]
NT$_{ADVP}$	[S, **NP**, VP, is, ADVP]	[still, ...]
SH	[S, **NP**, VP, is, ADVP, still]	[cautious, ...]
RE	[S, **NP**, VP, is, **ADVP**]	[cautious, ...]
NT$_{ADJP}$	[S, **NP**, VP, is, **ADVP**, ADJP]	[cautious, ...]
SH	[S, **NP**,VP, is, **ADVP**, ADJP, cautious]	[.]
RE	[S, **NP**, VP, is, **ADVP**, **ADJP**]	[.]
RE	[S, **NP**, **VP**]	[.]
SH	[S, **NP**, **VP**, **.**]	[ ]
RE	[**S**]	[ ]

Figure 3: Transition sequence for producing the constituent tree in Figure 1 using a top-down parser. SH = Shift, NT$_X$ = Non-Terminal-X and RE = Reduce. Already-built constituents are marked in bold.

We adapt the latter to efficiently implement a loss function for the top-down and in-order strategies.

While in bottom-up parsing constituents are created at once by a Reduce transition, in the other two approaches a Non-Terminal transition begins the process by naming the future constituent and a Reduce transition builds it by setting its span and children. Therefore, a Non-Terminal transition that deviates from the non-terminals expected in the gold tree will eventually produce a wrong constituent in future configurations, so it should be penalized. Additionally, a sequence of gold Non-Terminal transitions may also lead to a wrong final parse if they are applied in an incorrect order. Then, the computation of the Hamming loss in top-down and in-order phrase-structure parsing adds two more terms to the bottom-up loss expression: (1) the number of predicted non-terminal nodes that are currently in the stack (α_c),[3] but not included in the set of gold non-terminal nodes (α_G), and (2) the number of gold non-terminal nodes in the stack that are out of order with respect to the order needed in the gold tree:

$$\ell(c) = \min_{\gamma | c \rightsquigarrow \gamma} \mathcal{L}(\gamma, \gamma_G) = |\mathcal{U}(c, \gamma_G)| + |\gamma_c \setminus \gamma_G|$$

$$+ |\alpha_c \setminus \alpha_G| + out\_of\_order(\alpha_c, \alpha_G)$$

This loss function is used to implement a dynamic oracle that, when given any parser configuration, will return the set of transitions τ that do not increase the overall loss (i.e., $\ell(\tau(c)) - \ell(c) = 0$), leading the system through optimal configurations that minimize Hamming loss with respect to t_G.

As suggested by (Coavoux and Crabbé, 2016; Fernández-González and Gómez-Rodríguez, 2018), *constituent reachability* can be used to efficiently compute the first term of the symmetric difference ($|\gamma_G \setminus \gamma|$), by simply counting the gold constituents that are individually unreachable from configuration c, as we describe in the next subsection.

The second and third terms of the loss ($|\gamma_c \setminus \gamma_G|$ and $|\alpha_c \setminus \alpha_G|$) can be trivially computed and are used to penalize false positives (extra erroneous constituents) so that final F-score is not harmed due to the decrease of precision, as pointed out by (Coavoux and Crabbé, 2016; Fernández-González and Gómez-Rodríguez, 2018). Note that it is crucial that the creation of non-gold Non-Terminal transitions is avoided, since these might not affect the creation of gold constituents, however, they will certainly lead the parser to the creation of extra erroneous constituents in future steps.

Finally, the function out_of_order of the last term can be implemented by computing the *longest increasing subsequence* of gold non-terminal nodes in the stack, where the order relation is given by the order of non-terminals (provided by their associated index) in the transition sequence that builds the gold tree (this order is unique, as none of our two parsers of interest have spurious ambiguity). Obtaining the longest increasing subsequence is a well-known problem solvable in time $O(n\ log\ n)$ (Fredman, 1975), where n denotes the length of the input sequence. Once we have the largest possible sub-

[3]Note that we only consider predicted non-terminal nodes still in the stack, since wrong non-terminal nodes that have been already reduced are included in the loss as erroneous constituents.

Shift:	$\langle \Sigma, i, \mathit{false}, \gamma, \alpha \rangle \Rightarrow \langle \Sigma \vert (w_i, i, i+1), i+1, \mathit{false}, \gamma \cup \{(w_i, i, i+1)\}, \alpha \rangle$
Non-Terminal-X:	$\langle \Sigma, i, \mathit{false}, \gamma, \alpha \rangle \Rightarrow \langle \Sigma \vert (X, i), i, \mathit{false}, \gamma, \alpha \cup \{(X, i)\} \rangle$
Reduce:	$\langle \Sigma \vert (Y_1, m_0, m_1) \vert (X, j) \vert ... \vert (Y_k, m_{k-1}, m_k), i, \mathit{false}, \gamma, \alpha \rangle \Rightarrow \langle \Sigma \vert (X, m_0, m_k), i, \mathit{false}, \gamma \cup \{(X, m_0, m_k)\}, \alpha \setminus \{(X, j)\} \rangle$
Finish:	$\langle (S, 0, n), n, \mathit{false}, \gamma, \alpha \rangle \Rightarrow \langle (S, 0, n), n, \mathit{true}, \gamma, \alpha \rangle$

Figure 4: Transitions of a in-order constituent parser.

Transition	Σ	Buffer
	[ ]	[The, ...]
SH	[The]	[public, ...]
NT$_{\text{NP}}$	[The, NP]	[public, ...]
SH	[The, NP, public]	[is, ...]
RE	[**NP**]	[is, ...]
NT$_{\text{S}}$	[**NP**, S]	[is, ...]
SH	[**NP**, S, is]	[still, ...]
NT$_{\text{VP}}$	[**NP**, S, is, VP]	[still, ...]
SH	[**NP**, S, is, VP, still]	[cautious, ...]
NT$_{\text{ADVP}}$	[**NP**, S, is, VP, still, ADVP]	[cautious, ...]
RE	[**NP**, S, is, VP, **ADVP**]	[cautious, ...]
SH	[**NP**, S, is, VP, **ADVP**, cautious]	[.]
NT$_{\text{ADJP}}$	[**NP**, S, is,VP, **ADVP**, cautious, ADJP]	[.]
RE	[**NP**, S, is, VP, **ADVP**, **ADJP**]	[.]
RE	[**NP**, S, **VP**]	[.]
SH	[**NP**, S, **VP**, **.**]	[ ]
RE	[**S**]	[ ]
FI	[**S**]	[ ]

Figure 5: Transition sequence for producing the constituent tree in Figure 1 using an in-order parser. SH = Shift, NT$_{\text{X}}$ = Non-Terminal-X, RE = Reduce and FI = Finish. Already-built constituents are marked in bold.

sequence of gold non-terminal nodes in our configuration's stack that is compatible with the gold order, the remaining ones give us the number of erroneous constituents that we will unavoidably generate, even in the best case, due to building them in an incorrect order.

We will prove below that this loss formulation returns the exact loss and the resulting dynamic oracle is correct.

3.2 Constituent reachability

We now show how the computation of the set of reachable constituents developed for bottom-up parsing in (Coavoux and Crabbé, 2016; Fernández-González and Gómez-Rodríguez, 2018) can be extended to deal with the top-down and in-order strategies.

Top-down transition system Let γ_G and α_G be the set of gold constituents and the set of gold non-terminal nodes, respectively, for our current input. We say that a gold constituent $(X, l, r) \in \gamma_G$ is reachable from a configuration $c = \langle \Sigma, j, /, \gamma_c, \alpha_c \rangle$ with $\Sigma = [(Y_p, i_p, i_{p-1}) \cdots (Y_2, i_2, i_1) \vert (Y_1, i_1, j)]$, and it is included in the set of *individually reachable constituents* $\mathcal{R}(c, \gamma_G)$, iff it satisfies one of the following conditions:[4]

(i) $(X, l, r) \in \gamma_c$ (i.e. it has already been created and, therefore, it is reachable by definition).

(ii) $j \leq l < r \wedge (X, l) \notin \alpha_c$ (i.e. the words dominated by the gold constituent are still in the buffer and the non-terminal node that begins its creation has not been added to the stack yet; therefore, it can be still created after pushing the correct non-terminal node and shifting the necessary words).

(iii) $l \in \{i_k \mid 1 \leq k \leq p\} \wedge j \leq r \wedge (X, l) \in \alpha_c$ (i.e. its span is partially or completely in the stack and the corresponding non-terminal node was already added to the stack, then, by shifting more words or/and reducing, the constituent can still be created).

In-order transition system Let γ_G and α_G be the set of gold constituents and the set of gold non-terminal nodes, respectively, for our current input. We say that a gold constituent $(X, l, r) \in \gamma_G$ is reachable from a configuration $c = \langle \Sigma, j, \mathit{false}, \gamma_c, \alpha_c \rangle$ with $\Sigma = [(Y_p, i_p, i_{p-1}) \cdots (Y_2, i_2, i_1) \vert (Y_1, i_1, j)]$, and it is included in the set of *individually reachable constituents* $\mathcal{R}(c, \gamma_G)$, iff it satisfies one of the following conditions:

(i) $(X, l, r) \in \gamma_c$ (i.e. it has already been created).

(ii) $j \leq l < r$ (i.e. the constituent is entirely in the buffer, then it can be still built).

(iii) $l \in \{i_k \mid 1 \leq k \leq p\} \wedge j \leq r \wedge (X, m) \notin \alpha_c$ (i.e. its first child is still a totally- or partially-built constituent on top of the stack and the non-terminal node has not been created yet;

[4]Please note that elements from the stack can be an already-built constituent, a shifted word or a non-terminal node. Therefore, (Y_p, i_p, i_{p-1}), (Y_2, i_2, i_1) and (Y_1, i_1, j) should be represented as (Y_p, i_{p-1}), (Y_2, i_1) and (Y_1, j), respectively, when they are non-terminal nodes. We omit this for simplicity.

therefore, it has to wait till the first child is completed (if it is still pending) and, then, it can be still created by pushing onto the stack the correct non-terminal node and shifting more words if necessary).

(iv) $l \in \{i_k \mid 1 \leq k \leq p\} \wedge j \leq r \wedge (X, m) \in \alpha_c \wedge \exists (Y, l, m) \in \Sigma$ (i.e. its span is partially or completely in the stack, and its first child (which is an alredy-built constituent) and the non-terminal node assigned are adjacent, thus, by shifting more words or/and reducing, the constituent can still be built).

In both transition systems, the set of *individually unreachable constituents* $\mathcal{U}(c, \gamma_G)$ with respect to the set of gold constituents γ_G can be easily computed as $\gamma_G \setminus \mathcal{R}(c, \gamma_G)$ and will contain the gold constituents that can no longer be built.

3.3 Correctness

We will now prove that the above expression of $\ell(c)$ indeed provides the minimum possible Hamming loss to the gold tree among all the trees that are reachable from configuration c. This implies correctness (or optimality) of our oracle.

To do so, we first show that both algorithms are constituent-decomposable. This amounts to saying that if we take a set of m constituents that are tree-compatible (can appear together in a constituent tree, meaning that no pair of constituent spans overlap unless one is a subset of the other) and individually reachable from a configuration c, then the set is also reachable as a whole.

We prove this by induction on m. The base case ($m = 1$) is trivial. Let us suppose that constituent-decomposability holds for any set of m tree-compatible constituents. We will show that it also holds for any set T of $m+1$ tree-compatible constituents.

Let (X, l, r) be one of the constituents in T such that $r = \min\{r' \mid (X', l', r') \in T\}$ and $l = \max\{l' \mid (X', l', r) \in T\}$. Let $T' = T \setminus \{(X, l, r)\}$. Since T' has m constituents, by induction hypothesis, T' is a reachable set from configuration c.

Since (X, l, r) is individually reachable by hypothesis, it must satisfy at least one of the conditions for constituent reachability. As these conditions are different for each particular algorithm, we continue the proof separately for each:

Top-down constituent-decomposability In this case, we enumerated three constituent reachability conditions, so we divide the proof into three cases:

If the first condition holds, then the constituent (X, l, r) has already been created in c. Thus, it will still be present after applying any of the possible transition sequences that build T' starting from c. Hence, $T = T' \cup \{(X, l, r)\}$ is reachable from c.

If the second condition holds, then $j \leq l < r$ and the constituent (X, l, r) can be created by $l - j$ Shift transitions, followed by one Non-Terminal transition, $r - l$ Shift transitions and one Reduce transition. This will leave the parser in a configuration whose value of j is r, and where stack elements with left span index $\leq l$ (apart from those referencing the new non-terminal and its leftmost child) have not changed. Thus, constituents of T' are still individually reachable in this configuration, as their left span index is either $\geq r$ (and then they meet the second reachability condition) or $\leq l$ (and then they meet the third), so T is reachable from c.

Finally, if the third condition holds, then we can create (X, l, r) by applying $r - j$ Shift transitions followed by a sequence of Reduce transitions stopping when we obtain (X, l, r) on the stack (this will always happen after a finite number of such transitions, as the reachability condition guarantees that l is the left span index of some constituent already on the stack, and that (X, l) is on the stack). Following the same reasoning as in the previous case regarding the resulting parser configuration, we conclude that T is reachable from c.

With this we have shown the induction step, and thus constituent decomposability for the top-down parser.

In-order constituent decomposability The in-order parser has four constituent reachability conditions. Analogously to the previous case, we prove the reachability of T by case analysis.

If the first condition holds, then we have a situation where the constituent (X, l, r) has already been created in c, so reachability of T follows from the same reasoning as for the first condition in the top-down case.

If the second condition holds, we have $j \leq l < r$ and the constituent (X, l, r) can be created by $l - j + 1$ Shift transitions (where the last one shifts a word that will be assigned as left child of the new constituent), followed by the relevant Non-Terminal-X transition, $r - l - 1$ more Shift transitions and one Reduce transition. After this, the parser will be in a configuration where j takes the value r, where we can use the same reasoning as in the second condition of the top-down parser to show that all constituents of T' are still reachable, proving reachability of T.

For the third condition, the proof is analogous but the combination of transitions that creates the non-terminal starts with a sequence composed of Reduce transitions (when there is a non-terminal at the top of the stack) or Non-Terminal-Y transitions for arbitrary Y (when the top of the stack is a constituent) until the top node on the stack is a constituent with left span index l (this ensures that the constituent at the top of the stack can serve as leftmost child for our desired constituent), followed by a Non-Terminal-X, $r-j$ Shift transitions and one Reduce transition.

Finally, for the fourth condition, the reasoning is again analogous, but the computation leading to the non-terminal starts with as many Reduce transitions as non-terminal nodes located above (X, m) in the stack (if any). If we call j the index associated to the resulting transition, then it only remains to apply $r - j$ Shift transitions followed by a Reduce transition.

Optimality With this, we have shown constituent decomposability for both parsing algorithms. This means that, for a configuration c, and a set of constituents that are individually reachable from c, there is always some computation that can build them all. This facilitates the proof that the loss function is correct.

To finish the proof, we observe the following:

- Let c' be a final configuration reachable from c. The set $(\gamma_{c'} \setminus \gamma_G)$, representing erroneous constituents that have been built, will always contain at least $|\gamma_c \setminus \gamma_G|$, as the algorithm never deletes constituents.
- In addition, c' will contain one erroneous constituent for each element of $(\alpha_c \setminus \alpha_G)$, as once a non-terminal node is on the stack, there is no way to reach a final configuration without using it to create an erroneous constituent. Note that these erroneous constituents do not overlap those arising from the previous item, as γ_c stores already-built constituents and α_c non-terminals that have still not been used to build a constituent.
- Given a subset $\mathcal{S}$ of $\mathcal{R}(c, \gamma_G)$, the previously shown constituent decomposability property implies that there exists at least one transition sequence starting from c that generates the tree $\mathcal{S} \cup (\gamma_c \setminus \gamma_G) \cup E$, where E is a set of erroneous constituents containing one such constituent per element of $(\alpha_c \setminus \alpha_G)$. This tree has loss $|t_G| - (|\gamma_c \cup \mathcal{S}|) + |\gamma_c \setminus \gamma_G| + |\alpha_c \setminus \alpha_G|$. The term $|t_G| - (|\gamma_c \cup \mathcal{S}|)$ corresponds to missed constituents (gold constituents that have not been already created and are not created as part of $\mathcal{S}$), the other two to erroneous constituents.
- As we have shown that the erroneous constituents arising from $(\gamma_{c'} \setminus \gamma_G)$ and $(\alpha_c \setminus \alpha_G)$ are unavoidable, computations yielding a tree with minimum loss are those that maximize $|\gamma_c \cup \mathcal{S}|$ in the previous term. In general, the largest possible $|\mathcal{S}|$ is for $\mathcal{S} = \mathcal{R}(c, \gamma_G)$. In that case, we would correctly generate every reachable constituent and the loss would be

$$\ell(c) = |\mathcal{U}(c, \gamma_G)| + |\gamma_c \setminus \gamma_G| + |\alpha_c \setminus \alpha_G|$$

However, we additionally want to generate constituents in the correct order, and this may not be possible if we have already shifted some of them into the stack in a wrong order. The function out_of_order gives us the number of reachable constituents that are lost for this cause in the best case. Thus, indeed, the expression

$$\ell(c) = |\mathcal{U}(c, \gamma_G)| + |\gamma_c \setminus \gamma_G| + |\alpha_c \setminus \alpha_G| + out\_of\_order(\alpha_c, \alpha_G)$$

provides the minimum loss from configuration c.

4 Experiments

4.1 Data

We test the two proposed approaches on two widely-used benchmarks for constituent parsers: the Wall Street Journal (WSJ) sections of the English Penn Treebank[5] (Marcus et al., 1993) and version 5.1 of the Penn Chinese Treebank (CTB)[6] (Xue et al., 2005). We use the same predicted POS tags and pre-trained word embeddings as Dyer et al. (2016) and Liu and Zhang (2017a).

[5] Sections 2-21 are used as training data, Section 22 for development and Section 23 for testing

[6] Articles 001- 270 and 440-1151 are taken for training, articles 301-325 for system development, and articles 271-300 for final testing

4.2 Neural Model

To perform a fair comparison, we define the novel dynamic oracles on the original implementations of the top-down parser by Dyer et al. (2016) and in-order parser by Liu and Zhang (2017a), where parsers are trained with a traditional static oracle. Both implementations follow a stack-LSTM approach to represent the stack and the buffer, as well as a vanilla LSTM to represent the action history. In addition, they also use a bi-LSTM as a compositional function for representing constituents in the stack. Concretely, this consists in computing the composition representation s_{comp} as:

$$s_{comp} = (LSTM_{fwd}[e_{nt}, s_0, ..., s_m]; \\ LSTM_{bwd}[e_{nt}, s_m, ..., s_0])$$

where e_{nt} is the vector representation of a non-terminal, and s_i, $i \in [0, m]$ is the ith child node.

Finally, the exact same word representation strategy and hyper-parameter values as (Dyer et al., 2016) and (Liu and Zhang, 2017a) are used to conduct the experiments.

4.3 Error exploration

In order to benefit from training a parser by a dynamic oracle, errors should be made during the training process so that the parser can learn to avoid and recover from them. Unlike more complex error-exploration strategies as those studied in (Ballesteros et al., 2016; Cross and Huang, 2016b; Fried and Klein, 2018), we decided to consider a simple one that follows a non-optimal transition when it is the highest-scoring one, but with a certain probability. In that way, we easily simulate test time conditions, when the parser greedily chooses the highest-scoring transition, even when it is not an optimal one, placing the parser in an incorrect state.

In particular, we run experiments on development sets for each benchmark/algorithm with three different error exploration probabilities and choose the one that achieves the best F-score. Table 1 reports all results, including those obtained by the top-down and in-order parsers trained by a dynamic oracle without error exploration (equivalent to a traditional static oracle).

4.4 Results

Table 2 compares our system's accuracy to other state-of-the-art shift-reduce constituent parsers on the WSJ and CTB benchmarks. For comparison,

	Top-down		In-order	
Exp.	WSJ	CTB	WSJ	CTB
None	91.81	88.94	91.95	89.69
0.1	91.87	89.13	**92.05**	**89.91**
0.2	**91.99**	88.70	91.98	89.88
0.3	91.97	**89.20**	91.95	89.87

Table 1: F-score comparison of different error-exploration probabilities on WSJ §22 and CTB §301-325 for the top-down a in-order dynamic oracles.

Parser	Type	Strat	F1
(Cross and Huang, 2016a)	gs	bu	90.0
(Cross and Huang, 2016b)	gs	bu	91.0
(Cross and Huang, 2016b)	gd	bu	91.3
(Liu and Zhang, 2017a)	gs	bu	91.3
(Fernández-G and Gómez-R, 2018)	gs	bu	91.5
(Fernández-G and Gómez-R, 2018)	gd	bu	91.7
(Dyer et al., 2016)	gs	td	91.2
This work	gd	td	**91.7**
(Liu and Zhang, 2017a)	gs	in	91.8
This work	gd	in	**92.0**
(Zhu et al., 2013)	b	bu	90.4
(Watanabe and Sumita, 2015)	b	bu	90.7
(Liu and Zhang, 2017b)	b	bu	91.7
(Fried and Klein, 2018)	bp	td	91.6
(Fried and Klein, 2018)	bd	td	92.1
(Fried and Klein, 2018)	bp	in	92.2
(Stern et al., 2017b)	bg	td	92.6
(Stern et al., 2017a)	ch	bu	91.8
(Gaddy et al., 2018)	ch	bu	92.1
(Kitaev and Klein, 2018)	ch	bu	**93.6**

Parser	Type	Strat	F1
(Wang et al., 2015)	gs	bu	83.2
(Liu and Zhang, 2017a)	gs	bu	85.7
(Fernández-G and Gómez-R, 2018)	gs	bu	86.3
(Fernández-G and Gómez-R, 2018)	gd	bu	86.8
(Dyer et al., 2016)	gs	td	84.6
This work	gd	td	**85.3**
(Liu and Zhang, 2017a)	gs	in	86.1
This work	gd	in	**86.6**
(Zhu et al., 2013)	b	bu	83.2
(Watanabe and Sumita, 2015)	b	bu	84.3
(Liu and Zhang, 2017b)	b	bu	85.5
(Fried and Klein, 2018)	bd	td	85.5
(Fried and Klein, 2018)	bp	td	84.7
(Fried and Klein, 2018)	bp	in	**87.0**

Table 2: Accuracy comparison of state-of-the-art single-model fully-supervised constituent parsers on WSJ §23 (top) and CTB §271-300 (bottom). The "Type" column shows the type of parser: *gs* is a greedy parser trained with a static oracle, *gd* a greedy parser trained with a dynamic oracle, *b* a beam search parser, *bp* a beam search parser trained with a policy gradient method, *bd* a beam search parser trained with a non-optimal dynamic oracle, *bg* a generative beam search parser, and *ch* a chart-based parser. Finally, the "Strat" column describes the strategy followed (*bu*=bottom-up, *td*=top-down and *in*=in-order).

Parser	Oracle	#1	#2	#3	#4	#5
Top-down	static	90.98	88.76	85.01	76.63	77.35
	dynamic	**91.34** (+0.36)	**89.18** (+0.42)	**85.17** (+0.16)	**77.12** (+0.49)	**80.02** (+2.67)
In-order	static	91.36	89.21	85.15	77.08	79.02
	dynamic	**91.55** (+0.19)	**89.43** (+0.22)	**85.34** (+0.19)	**77.57** (+0.49)	**81.03** (+2.01)

Table 3: F-score on constituents with a number of children ranging from one to five on WSJ §23.

we also include some recent state-of-the-art parsers with global chart decoding that achieve the highest accuracies to date on WSJ, but are much slower than shift-reduce algorithms.

Top-down and in-order parsers benefit from being trained by these new dynamic oracles in both datasets. The top-down strategy achieves a gain of 0.5 and 0.7 points in F-score on WSJ and CTB benchmarks, respectively. The in-order parser obtains similar improvements on the CTB (0.5 points), but less notable accuracy gain on the WSJ (0.2 points). Although a case of diminishing returns might explain the latter, the in-order parser trained with the proposed dynamic oracle still achieves the highest accuracy to date in greedy transition-based constituent parsing on the WSJ.[7]

While this work was under review, Fried and Klein (2018) proposed to train the top-down and in-order parsers with a policy gradient method instead of custom designed dynamic oracles. They also present a non-optimal dynamic oracle for the top-down parser that, combined with more complex error-exploration strategies and size-10 beam search, significantly outperforms the policy gradient-trained version, confirming that even non-optimal dynamic oracles are a good option.[8]

4.5 Analysis

Dan Bikel's randomized parsing evaluation comparator (Bikel, 2004) was used to perform significance tests on precision and recall metrics on WSJ §23 and CTB §271-300. The top-down parser trained with dynamic oracles achieves statistically significant improvements ($p < 0.05$) in precision both on the WSJ and CTB benchmarks, and in recall on WSJ. The in-order parser trained with the proposed technique obtains significant improvements ($p < 0.05$) in recall in both benchmarks, although not in precision.

We also undertake an analysis to check if dynamic oracles are able to mitigate error propagation. We report in Table 3 the F-score obtained in constituents with different number of children on WSJ §23 by the top-down and in-order algorithms trained with both static and dynamic oracles. Please note that creating a constituent with a great number of children is more prone to suffer from error propagation, since a larger number of transitions is required to build it. The results seem to confirm that, indeed, dynamic oracles manage to alleviate error propagation, since improvements in F-score are more notable for larger constituents.

5 Conclusion

We develop the first optimal dynamic oracles for training the top-down and the state-of-the-art in-order parsers. Apart from improving the systems' accuracies in both cases, we achieve the best result to date in greedy shift-reduce parsing on the WSJ. In addition, these promising techniques could easily benefit from recent studies in error-exploration strategies and yield state-of-the-art accuracies in transition-based parsing in the near future. The parser's source code is freely available at `https://github.com/danifg/Dynamic-InOrderParser`.

Acknowledgments

This work has received funding from the European Research Council (ERC), under the European Union's Horizon 2020 research and innovation programme (FASTPARSE, grant agreement No 714150), from MINECO (FFI2014-51978-C2-2-R, TIN2017-85160-C2-1-R) and from Xunta de Galicia (ED431B 2017/01).

[7]Note that the proposed dynamic oracles are orthogonal to approaches like beam search, re-ranking or semi-supervision, that can boost accuracy but at a large cost to parsing speed.

[8]Unfortunately, we cannot directly compare our approach to theirs, since they use beam-search decoding with size 10 in all experiments, gaining up to 0.3 points in F-score, while penalizing speed with respect to greedy decoding. However, by extrapolating the results above, we hypothesize that our optimal dynamic oracles (especially the one designed for the in-order algorithm) with their same training and beam-search decoding setup might achieve the best scores to date in shift-reduce parsing.

References

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A. Smith. 2016. Training with exploration improves a greedy stack LSTM parser. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2005–2010.

Dan Bikel. 2004. *On the Parameter Space of Generative Lexicalized Statistical Parsing Models*. Ph.D. thesis, University of Pennsylvania.

Maximin Coavoux and Benoit Crabbé. 2016. Neural greedy constituent parsing with dynamic oracles. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 172–182, Berlin, Germany. Association for Computational Linguistics.

Benoit Crabbé. 2015. Multilingual discriminative lexicalized phrase structure parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1847–1856, Lisbon, Portugal. Association for Computational Linguistics.

James Cross and Liang Huang. 2016a. Incremental parsing with minimal features using bi-directional LSTM. In *ACL (2)*. The Association for Computer Linguistics.

James Cross and Liang Huang. 2016b. Span-based constituency parsing with a structure-label system and provably optimal dynamic oracles. In *EMNLP*, pages 1–11. The Association for Computational Linguistics.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *HLT-NAACL*, pages 199–209. The Association for Computational Linguistics.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2018. Faster shift-reduce constituent parsing with a non-binary, bottom-up strategy. *arXiv*, 1804.07961 [cs.CL].

Michael L. Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29 – 35.

Daniel Fried and Dan Klein. 2018. Policy gradient as a proxy for dynamic oracles in constituency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*, pages 469–476.

David Gaddy, Mitchell Stern, and Dan Klein. 2018. What's going on in neural constituency parsers? an analysis. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 999–1010.

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

Nikita Kitaev and Dan Klein. 2018. Constituency parsing with a self-attentive encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 2675–2685.

Adhiguna Kuncoro, Miguel Ballesteros, Lingpeng Kong, Chris Dyer, Graham Neubig, and Noah A. Smith. 2017. What do recurrent neural network grammars learn about syntax? In *EACL (1)*, pages 1249–1258. Association for Computational Linguistics.

Jiangming Liu and Yue Zhang. 2017a. In-order transition-based constituent parsing. *Transactions of the Association for Computational Linguistics*, 5:413–424.

Jiangming Liu and Yue Zhang. 2017b. Shift-reduce constituent parsing with neural lookahead features. *TACL*, 5:45–58.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330.

Haitao Mi and Liang Huang. 2015. Shift-reduce constituency parsing with dynamic programming and pos tag lattice. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1030–1035, Denver, Colorado. Association for Computational Linguistics.

Kenji Sagae and Alon Lavie. 2005. A classifier-based parser with linear run-time complexity. In *Proceedings of the 9th International Workshop on Parsing Technologies (IWPT)*, pages 125–132.

Mitchell Stern, Jacob Andreas, and Dan Klein. 2017a. A minimal span-based neural constituency parser. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 818–827.

Mitchell Stern, Daniel Fried, and Dan Klein. 2017b. Effective inference for generative neural parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1695–1700.

Zhiguo Wang, Haitao Mi, and Nianwen Xue. 2015. Feature optimization for constituent parsing via neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1138–1147.

Taro Watanabe and Eiichiro Sumita. 2015. Transition-based neural constituent parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics, ACL 2015, 26-31 July 2015, Bejing, China, Volume 1: Long Papers*, pages 1169–1179.

Naiwen Xue, Fei Xia, Fu-dong Chiou, and Marta Palmer. 2005. The penn chinese treebank: Phrase structure annotation of a large corpus. *Nat. Lang. Eng.*, 11(2):207–238.

Yue Zhang and Stephen Clark. 2009. Transition-based parsing of the chinese treebank using a global discriminative model. In *Proceedings of the 11th International Conference on Parsing Technologies*, IWPT '09, pages 162–171, Stroudsburg, PA, USA. Association for Computational Linguistics.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 434–443.