Discontinuous Constituency Parsing with a Stack-Free Transition System and a Dynamic Oracle

Citation for published version:

Digital Object Identifier (DOI):
10.18653/v1/n19-1018

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Discontinuous Constituency Parsing with a Stack-Free Transition System and a Dynamic Oracle

Maximin Coavoux
Naver Labs Europe
maximin.coavoux@naverlabs.com

Shay B. Cohen
ILCC, School of Informatics
University of Edinburgh
scohen@inf.ed.ac.uk

Abstract

We introduce a novel transition system for discontinuous constituency parsing. Instead of storing subtrees in a stack—i.e. a data structure with linear-time sequential access—the proposed system uses a set of parsing items, with constant-time random access. This change makes it possible to construct any discontinuous constituency tree in exactly \(4n-2\) transitions for a sentence of length \(n\), whereas existing systems need a quadratic number of transitions to derive some structures. At each parsing step, the parser considers every item in the set to be combined with a focus item and to construct a new constituent in a bottom-up fashion. The parsing strategy is based on the assumption that most syntactic structures can be parsed incrementally and that the set—the memory of the parser—remains reasonably small on average. Moreover, we introduce a dynamic oracle for the new transition system, and present the first experiments in discontinuous constituency parsing using a dynamic oracle. Our parser obtains state-of-the-art results on three English and German discontinuous treebanks.

1 Introduction

Discontinuous constituency trees extend standard constituency trees by allowing crossing branches to represent long distance dependencies, such as the \(wh\)-extraction in Figure 1. Discontinuous constituency trees can be seen as derivations of Linear Context-Free Rewriting Systems (LCFRS, Vijay-Shanker et al., 1987), a class of formal grammars more expressive than context-free grammars, which makes them much harder to parse. In particular, exact CKY-style LCFRS parsing has an \(O(n^{3f})\) time complexity where \(f\) is the fan-out of the grammar (Kallmeyer, 2010).

A natural alternative to grammar-based chart parsing is transition-based parsing, that usually relies on fast approximate decoding methods such as greedy search or beam search. Transition-based discontinuous parsers construct discontinuous constituents by reordering terminals with the \textit{SWAP} action (Versley, 2014a,b; Maier, 2015; Maier and Lichte, 2016; Stanojević and Garrido Alhama, 2017), or by using a split stack and the \textit{GAP} action to combine two non-adjacent constituents (Coavoux and Crabbé, 2017a; Coavoux et al., 2019). These proposals represent the memory of the parser (i.e. the tree fragments being constructed) with data structures with \textbf{linear-time sequential access} (either a stack, or a stack coupled with a double-ended queue). As a result, these systems need to perform at least \(n\) actions to construct a new constituent from two subtrees separated by \(n\) intervening subtrees. Our proposal aims at avoiding this cost when constructing discontinuous constituents.

We design a novel transition system in which a discontinuous constituent is constructed in a single step, without the use of reordering actions such as \textit{SWAP}. The main innovation is that the memory of the parser is not represented by a stack,
that derives any tree in exactly \(4n - 2\) steps for a sentence of length \(n\):

- we introduce the first dynamic oracle for discontinuous constituency parsing;
- we present an empirical evaluation of the transition system and dynamic oracle on two German and one English discontinuous tree-banks.

The code of our parser is released as an open-source project at https://gitlab.com/mcoavoux/discoparse.

## 2 Set-based Transition System

### System overview

We propose to represent the memory of the parser by (i) a set of parsing items and (ii) a single focus item. Figure 2 (lower part) illustrates a configuration in our system. The parser constructs a tree with two main actions: shift the next token to make it the new focus item (SHIFT), or combine any item in the set with the focus item to make a new constituent bottom-up (COMBINE action).

Since the memory is not an ordered data structure, the parser considers equally every pending parsing item, and thus constructs a discontinuous constituent in a single step, thereby making it able to construct any discontinuous tree in \(O(n)\) transitions.

The use of an unordered random-access data structure to represent the memory of the parser also leads to a major change for the scoring system (Figure 2). Stack-based systems use a local view of a parsing configuration to extract features and score actions: features only rely on the few top-most elements on the stack and buffer. The score of each transition depends on the totality of this local view. In contrast, we consider equally every item in the set, and therefore rely on a global view.

### 2.1 Set-based transition system

<table>
<thead>
<tr>
<th>Structural actions</th>
<th>Input</th>
<th>Output</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIFT</td>
<td>((S, s_f, i, C): j \Rightarrow (S \cup {s_f}, {i}, i + 1, C): j + 1)</td>
<td>(i &lt; n, j) is even</td>
<td></td>
</tr>
<tr>
<td>COMBINE-s</td>
<td>((S, s_f, i, C): j \Rightarrow (S - s, s_f \cup s, i, C): j + 1)</td>
<td>(s \in S, j) is even</td>
<td></td>
</tr>
</tbody>
</table>

### 2.2 Labelling actions

<table>
<thead>
<tr>
<th>Labelling actions</th>
<th>Input</th>
<th>Output</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL-X</td>
<td>((S, s_f, i, C): j \Rightarrow (S, s_f, i, C \cup {X, s_f}): j + 1)</td>
<td>(j) is odd</td>
<td></td>
</tr>
<tr>
<td>NO-LABEL</td>
<td>((S, s_f, i, C): j \Rightarrow (S, s_f, i, C): j + 1)</td>
<td>(i \neq n) or (S \neq \emptyset), (j) is odd</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Set-based transition system description. Variable \(j\) is the number of steps performed since the start of the derivation.
are respectively represented by \((VP, right-index based on the following types of actions:

\[ min(|S|) \times \max(|N|) + 1 \]

...transitions are divided into structural actions (SHIFT, COMBINE-s) and labelling actions (LABEL-X, NO-LABEL). The parser may only perform a structural action on an even step and a labelling action on an odd step. For our system, this distinction has the crucial advantage of keeping the number of possible actions low at each parsing step, compared to a system that would perform a COMBINE action and a labelling action in a single REDUCE-s-X action.

Table 1 presents each action as a deduction rule associated with preconditions. In Table 2, we describe how to derive the tree from Figure 1.

2.2 Oracles

Training a transition-based parser requires an oracle, i.e. a function that determines what the best action is in a specific parsing configuration to serve as a training signal. We first describe a static oracle that provides a canonical derivation for a given gold tree. We then introduce a dynamic oracle that determines what the best action is in any parsing configuration.

---

1In such a case, we would need to score \(|S| \times |N| + 1\) actions, where \(N\) is the set of nonterminals, instead of \(|S| + 1\) actions for our system.
2.2.1 Static Oracle

Our transition system exhibits a fair amount of 
spurious ambiguity, the ambiguity exhibited by 
the existence of many possible derivations for a 
single tree. Indeed, since we use an unordered 
memory, an n-ary constituent (and more generally 
a tree) can be constructed by many different tran-
sition sequences. For example, the set 
\{0, 1, 2\} might be constructed by combining 
- \{0\} and \{1\} first, and the result with \{2\}; or 
- \{1\} and \{2\} first, and the result with \{0\}; or 
- \{0\} and \{2\} first, and the result with \{1\}.

Following Cohen et al. (2012), we elimi-
nate spurious ambiguity by selecting a canonical 
derivation for a gold tree. In particular, we design 
the static oracle (i) to apply COMBINE as soon as 
possible in order to minimize the size of the mem-
ory (ii) to combine preferably with the most recent 
set in the memory when several combinations are 
possible. The first choice is motivated by proper-
ties of our system: when the memory is smaller, 
there are fewer choices, therefore decisions are 
simpler and less expensive to score.

2.2.2 Dynamic Oracle

Parsers are usually trained to predict the gold se-
quence of actions, using a static oracle. The lim-
itation of this method is that the parser only sees 
a tiny portion of the search space at train time and 
only trains on gold input (i.e. configurations ob-
tained after performing gold actions). At test time, 
it is in a different situation due to error propaga-
tion: it must predict what the best actions are in 
configurations from which the gold tree is proba-
bly no longer reachable.

To alleviate this limitation, Goldberg and Nivre 
(2012) proposed to train a parser with a dynamic 
oracle, an oracle that is defined for any parsing 
configuration and outputs the set of best actions to 
perform. In contrast, a static oracle is determinis-
tic and is only defined for gold configurations.

Dynamic oracles were proposed for a wide 
range of dependency parsing transition systems 
(Goldberg and Nivre, 2013; Gómez-Rodríguez et al., 
2014; Gómez-Rodríguez and Fernández-
González, 2015), and later adapted to constituency 
parsing (Coavoux and Crabbé, 2016; Cross and 
Huang, 2016b; Fernández-González and Gómez-
Rodríguez, 2018b,a).

In the remainder of this section, we introduce 
a dynamic oracle for our proposed transition sys-
tem. It can be seen as an extension of the oracle of 
Cross and Huang (2016b) to the case of disconti-
uuous parsing.

Preliminary definitions. For a parsing configu-
ration \(c\), the relation \(c \vdash c'\) holds iff \(c'\) can be 
derived from \(c\) by a single transition. We note 
\(\vdash^*\) the reflexive and transitive closure of \(\vdash\). An 
instantiated constituent \((X, s)\) is reachable 
from a configuration \(c = (S, s_f, i, C)\) iff there exists 
\(c' = (S', s'_f, i', C')\) such that \((X, s) \in C'\) and 
\(c \vdash^* c'\). Similarly, a set of constituents \(t\) (possibly 
a full discontinuous constituency tree) is reachable 
iff there exists a configuration \(c' = (S', s'_f, i', C')\) 
such that \(t \subseteq C'\) and \(c \vdash^* c'\). We note 
\(\text{reach}(c, t^*)\) the set of constituents that are 
(i) in the gold set of constituents \(t^*\) (ii) reachable from \(c\).

We define a total order \(\preceq\) on index sets:

\[
s \preceq s' \iff \begin{cases} 
\max(s) < \max(s'), & \text{or} \\
\max(s) = \max(s'), \text{ and } s \subseteq s'. 
\end{cases}
\]

This order naturally extends to the constituents of 
a tree: \((X, s) \preceq (X', s')\) iff \(s \preceq s'\). If \((X, s)\) 
precedes \((X', s')\), then \((X, s)\) must be constructed 
before \((X', s')\). Indeed, since the right-index of 
the focus item is non-decreasing during a deriva-
tion (as per the transition definitions), constituents 
are constructed in the order of their right-index 
(first condition). Moreover, since the algorithm is 
bottom-up, a constituent must be constructed be-
fore its parent (second condition).

From a configuration \(c = (S, s_f, i, C)\) at an odd 
step, a constituent \((X, s_g)\) \(\notin C\) is reachable iff 
both the following properties hold:

1. \(\max(s_f) \leq \max(s_g)\);
2. \(\forall s \in S \cup \{s_f\}, (s \subseteq s_g)\) or \((s \cap s_g = \emptyset)\).

Condition 1 is necessary because the parser can 
only construct new constituents \((X, s)\) such that 
\(s_f \preceq s\). Condition 2 makes sure that \(s_g\) 
can be constructed from a union of elements from 
\(S \cup \{s_f\}\), potentially augmented with terminals from 
the buffer: \(\{i, i + 1, \ldots, \max(s_g)\}\).

Following Cross and Huang (2016b), we de-
fine \(\text{next}(c, t^*)\) as the smallest reachable gold con-
stituent from a configuration \(c\). Formally:

\[
\text{next}(c, t^*) = \arg\min_{t'} \text{reach}(c, t^*). 
\]
**Oracle algorithm** We first define the oracle 0 for the odd step of a configuration \( c = (S, sf, i, C) \):

\[
0_{\text{odd}}(c, t^*) = \begin{cases} 
\{|\text{LABEL-X}\} & \text{if } \exists (X, sf) \in t^*, \\
\{|\text{NO-LABEL}\} & \text{otherwise.}
\end{cases}
\]

For even steps, assuming \( \text{next}(c, t^*) = (X, sg) \), we define the oracle as follows:

\[
0_{\text{even}}(c, t^*) = \begin{cases} 
\{|\text{COMB-s}|(sf \cup s) \subseteq sg\} \text{ if } \max(sg) = \max(sf), \\
\{|\text{COMB-s}|(sf \cup s) \subseteq sg\} \cup \{|\text{SH}\} & \text{if } \max(sg) > \max(sf).
\end{cases}
\]

We provide a proof of the correctness of the oracle in Appendix A.

### 3 A Neural Network based on Constituent Boundaries

We first present an encoder that computes context-aware representations of tokens (Section 3.1). We then discuss how to compute the representation of a set of tokens (Section 3.2). We describe the action scorer (Section 3.3), the POS tagging component (Section 3.4), and the objective function (Section 3.5).

#### 3.1 Token Representations

As in recent proposals in dependency and constituency parsing (Cross and Huang, 2016a; Kiperwasser and Goldberg, 2016), our scoring system is based on a sentence transducer that constructs a context-aware representation for each token.

Given a sequence of tokens \( x^n_1 = (x_1, \ldots, x_n) \), we first run a single-layer character bi-LSTM encoder \( e \) to obtain a character-aware embedding \( e(x_i) \) for each token. We represent a token \( x_i \) as the concatenation of a standard word embedding \( w_{x_i} = [e(x_i); e(x_i)] \).

Then, we run a 2-layer bi-LSTM transducer over the sequence of token representations:

\[
(h_i^{(1)}, \ldots, h_i^{(n)}) = \text{bi-LSTM}(w_{x_1}, \ldots, w_{x_n}),
\]

\[
(h_i^{(2)}, \ldots, h_i^{(2)}) = \text{bi-LSTM}(h_i^{(1)}, \ldots, h_i^{(1)}).
\]

The parser uses the context-aware token representations \( h_i^{(2)} \) to construct vector representations of sets or constituents.

#### 3.2 Set Representations

An open issue in neural discontinuous parsing is the representation of discontinuous constituents. In projective constituency parsing, it has become standard to use the boundaries of constituents (Hall et al., 2014; Crabbé, 2015; Durret and Klein, 2015), an approach that proved very successful with bi-LSTM token representations (Cross and Huang, 2016b; Stern et al., 2017).

Although constituent boundary features improve discontinuous parsing (Coavoux and Crabbé, 2017a), relying only on the left-index and the right-index of a constituent has the limitation of ignoring gaps inside a constituent. For example, since the two VPs in Figure 1 have the same right-index and left-index, they would have the same representations. It may also happen that constituents with identical right-index and left-index do not have the same labels.

We represent a (possibly partial) constituent with the yield \( s \), by computing 4 indexes from \( s \): \( (\min(s), \max(s), \min(\pi), \max(\pi)) \). The set \( \pi \) represents the gap in \( s \), i.e. the tokens between \( \min(s) \) and \( \max(s) \) that are not in the yield of \( s \):

\[
\pi = \{ i | \min(s) < i < \max(s) \text{ and } i \notin s \}.
\]

Finally, we extract the corresponding token representations of the 4 indexes and concatenate them to form the vector representation \( r(s) \) of \( s \):

\[
r(s) = [h_{\min(s)}^{(2)}; h_{\max(s)}^{(2)}; h_{\min(\pi)}^{(2)}; h_{\max(\pi)}^{(2)}].
\]

For an index set that does not contain a gap, we have \( \pi = \emptyset \). To handle this case, we use a parameter vector \( h_{\text{nil}} \), randomly initialized and learned jointly with the network, to embed \( \max(\emptyset) = \min(\emptyset) = \text{nil} \).

For example, the constituents \( \text{VP, \{1, 6\}} \) and \( \text{VP, \{1, 5, 6\}} \) will be respectively vectorized as:

\[
r(\{1, 6\}) = [h_1^{(2)}; h_6^{(2)}; h_2^{(2)}; h_5^{(2)}],
\]

\[
r(\{1, 5, 6\}) = [h_1^{(2)}; h_6^{(2)}; h_2^{(2)}; h_4^{(2)}] .
\]

This representation method makes sure that two distinct index sets have distinct representations, as long as they have at most one gap each. This property no longer holds if one index sets has more than one gap.

#### 3.3 Action Scorer

For each type of action—structural or labelling—we use a feedforward network with two hidden layers.
Structural actions  At structural steps, for a configuration \( c = (S, s_f, i, C) \), we need to compute the score of \(|S| \) COMBINE actions and possibly a SHIFT action. In our approach, the score of a combine-s action only depends on \( s \) and \( s_f \) and is independent of the rest of the configuration (i.e. other items in the set). We first construct input matrix \( M \) as follows:

\[
M = \begin{pmatrix}
    r(s_1) & \cdots & r(s_n) & r\{i\}
    \\
    r(s_f) & \cdots & r(s_f) & r(s_f)
\end{pmatrix}.
\]

Each of the first \( n \) columns of matrix \( M \) represents the input for a COMBINE action, whereas the last column is the input for the SHIFT action. We then compute the score of each structural action:

\[
P(\cdot | c) = \text{Softmax}(\text{FF}_s(M)),
\]

where \( \text{FF}_s \) is a feedforward network with two hidden layers, a tanh activation and a single output unit. In other words, it outputs a single scalar for each column vector of matrix \( M \). This part of the network can be seen as an attention mechanism, where the focus item is the query, and the context is formed by the items in the set and the first element in the buffer.

Labelling actions  We compute the probabilities of labelling actions as follows:

\[
P(\cdot | s_f) = \text{Softmax}(\text{FF}_l(r(s_f))),
\]

where \( \text{FF}_l \) is a feedforward network with two hidden layers activated with the tanh function, and \(|N| + 1 \) output units, where \( N \) is the set of nonterminals.

3.4 POS Tagger

Following Coavoux and Crabbé (2017b), we use the first layer of the bi-LSTM transducer as input to a Part-of-Speech (POS) tagger that is learned jointly with the parser. For a sentence \( x^n \), we compute the probability of a sequence of POS tags \( t^n \) as follows:

\[
P(t^n | x^n) = \prod_{i=1}^{n} \text{Softmax}(W^{(i)} \cdot h^{(i)} + b^{(i)})_{t_i},
\]

where \( W^{(i)} \) and \( b^{(i)} \) are parameters.

3.5 Objective Function

In the static oracle setting, for a single sentence \( x^n \), we optimize the sum of the log-likelihood of gold POS-tags \( t^n \) and the log-likelihood of gold parsing actions \( a^n \):

\[
L = L_t + L_p,
\]

\[
L_t = -\sum_{i=1}^{n} \log P(t_i | x^n),
\]

\[
L_p = -\sum_{i=1}^{n} \log P(a_i | a^{i-1}, x^n).
\]

We optimize this objective by alternating a stochastic step for the tagging objective and a stochastic step for the parsing objective, as is standard in multitask learning (Caruana, 1997).

In the dynamic oracle setting, instead of optimizing the likelihood of the gold actions (assuming all previous actions were gold), we optimize the likelihood of the best actions, as computed by the dynamic oracle, from a configuration sampled from the space of all possible configurations. In practice, before each epoch, we sample each sentence from the training corpus with probability \( p \) and we use the current (non-averaged) parameters to parse the sentence and generate a sequence of configurations. Instead of selecting the highest-scoring action at each parsing step, as in a normal inference step, we sample an action using the softmax distribution computed by the parser, as done by Ballesteros et al. (2016). Then, we use the dynamic oracle to calculate the best action from each of these configurations. In case there are several best actions, we deterministically choose a single action by favoring a COMBINE over a SHIFT (to bias the model towards a small memory), and to COMBINE with the item with the highest right-index (to avoid spurious discontinuity in partial constituents). We train the parser on these sequences of potentially non-gold configuration-action pairs.

4 Experiments

We carried out experiments to assess the adequacy of our system and the effect of training with the dynamic oracle. We present the three discontinuous constituency treebanks that we used (Section 4.1), our experimental protocol (Section 4.2), then we discuss the results (Section 4.3) and the efficiency of the parser (Section 4.4).
Table 3: Results on development corpora. F1 is the Fscore on all constituents, Disc. F1 is an Fscore computed only on discontinuous constituents, POS is the accuracy on part-of-speech tags. Detailed results (including precision and recall) are given in Table 7 of Appendix C.

<table>
<thead>
<tr>
<th>Model</th>
<th>English (DPTB)</th>
<th>German (Tiger)</th>
<th>German (Negra)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1</td>
<td>Disc. F1</td>
<td>F1</td>
</tr>
<tr>
<td>This work, dynamic oracle</td>
<td>90.9</td>
<td>67.3</td>
<td>82.5</td>
</tr>
<tr>
<td>Coavoux et al. (2019),* GAP, bi-LSTM</td>
<td>91.0</td>
<td>71.3</td>
<td>82.7</td>
</tr>
<tr>
<td>Stanojević and Garrido Alhama (2017),* SWAP, stack/tree-LSTM</td>
<td>77.0</td>
<td>79.3</td>
<td>79.5</td>
</tr>
<tr>
<td>Coavoux and Crabbé (2017a), SR-GAP, perceptron</td>
<td>79.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versley (2016), pseudo-projective, chart-based</td>
<td>89.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corro et al. (2017),* bi-LSTM, Maximum Spanning Arborescence</td>
<td>87.0</td>
<td>74.8</td>
<td></td>
</tr>
<tr>
<td>van Cranenburgh et al. (2016), DOP, ≤ 40</td>
<td>77.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fernández-González and Martins (2015), dependency-based</td>
<td>75.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gebhardt (2018), LCFRS with latent annotations</td>
<td>90.1</td>
<td>81.6</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Discontinuous parsing results on the test sets.
*Neural scoring system. †Does not discount root symbols and punctuation.

4.1 Datasets

We perform experiments on three discontinuous constituency corpora. The discontinuous Penn Treebank was introduced by Evang and Kallmeyer (2011) who converted the long distance dependencies encoded by indexed traces in the original Penn treebank (Marcus et al., 1993) to discontinuous constituents. We used the standard split (sections 2-21 for training, 22 for development and 23 for test). The Tiger corpus (Brants et al., 2004) and the Negra corpus (Skut et al., 1997) are both German treebanks natively annotated with discontinuous constituents. We used the SPMRL split for the Tiger corpus (Seddah et al., 2013), and the split of Dubey and Keller (2003) for the Negra corpus.

4.2 Implementation and Protocol

We implemented our parser in Python using the Pytorch library (Paszke et al., 2017). We trained each model with the ASGD algorithm (Polyak and Juditsky, 1992) for 100 epochs. Training a single model takes approximately a week with a GPU. We evaluate a model every 4 epochs on the validation set and select the best performing model according to the validation F-score. We refer the reader to Table 5 of Appendix B for the full list of hyperparameters.

We evaluate models with the dedicated module of discodop² (van Cranenburgh et al., 2016). We use the standard evaluation parameters (proper.prm), that ignore punctuations and root symbols. We report two evaluation metrics: a standard Fscore (F) and an Fscore computed only on discontinuous constituents (Disc. F), which provides a more qualitative evaluation of the ability of the parser to recover long distance dependencies.

²https://github.com/andreasvc/disco-dop
4.3 Results

Effect of Dynamic Oracle  We present parsing results on the development sets of each corpus in Table 3. The effect of the oracle is in line with other published results in projective constituency parsing (Coavoux and Crabbé, 2016; Cross and Huang, 2016b) and dependency parsing (Goldberg and Nivre, 2012; Gómez-Rodríguez et al., 2014): the dynamic oracle improves the generalization capability of the parser.

External comparisons  In Table 4, we compare our parser to other transition-based parsers (Maier, 2015; Coavoux and Crabbé, 2017a; Stanojević and Garrido Alhama, 2017; Coavoux et al., 2019), the pseudo-projective parser of Versley (2016), grammar-based chart parsers (Evang and Kallmeyer, 2011; van Cranenburgh et al., 2016; Gebhardt, 2018) and parsers based on dependency parsing (Fernández-González and Martins, 2015; Corro et al., 2017). Note that some of them only report results in a gold POS tag setting (the parser has access to gold POS tags and use them as features), a setting that is much easier than ours.

Our parser matches the state of the art of Coavoux et al. (2019). This promising result shows that it is feasible to design accurate transition systems without an ordered memory.

4.4 Efficiency

Our transition system derives a tree for a sentence of $n$ words in exactly $4n - 2$ transitions. Indeed, there must be $n$ SHIFT actions, and $n-1$ COMBINE actions. Each of these $2n-1$ transitions must be followed by a single labelling action.

The statistical model responsible for choosing which action to perform at each parsing step needs to score $|S| + 1$ actions for a structural step and $|N| + 1$ actions for a labelling step (where $N$ is the set of possible nonterminals). Since in the worst case, $|S|$ contains $n-1$ singletons, the parser has an $O(n(|N| + n))$ time complexity.

In practice, the memory of the parser $S$ remains relatively small on average. We report in Figure 3 the distribution of the size of $S$ across configurations when parsing the development set of the three corpora. For the German treebanks, the memory contains 7 or fewer elements for more than 99 percent of configurations. For the Penn treebank, the memory is slightly larger, with 98 percent of configuration with 11 or fewer items.

We report empirical runtimes in Table 6 of Appendix C. Our parser compares decently with other transition-based parsers, despite being written in Python.

5 Related Work

Existing transition systems for discontinuous constituency parsing rely on three main strategies for constructing discontinuous constituents: a swap-
based strategy, a split-stack strategy, and the use of non-local transitions.

**Swap-based systems** Swap-based transition systems are based on the idea that any discontinuous constituency tree can be transformed into a projective tree by reordering terminals. They reorder terminals by swapping them with a dedicated action (SWAP), commonly used in dependency parsing (Nivre, 2009). The first proposals in transition-based discontinuous constituency parsing used the SWAP action on top of an easy-first parser (Versley, 2014a,b). Subsequent proposals relied on a shift-reduce system (Maier, 2015; Maier and Lichte, 2016) or a shift-promote-adjoin system (Stanojević and Garrido Alhama, 2017).

The main limitation of swap-based system is that they tend to require a large number of transitions to derive certain trees. The choice of an oracle that minimizes derivation lengths has a substantially positive effect on parsing (Maier and Lichte, 2016; Stanojević and Garrido Alhama, 2017).

**Split-stack systems** The second parsing strategy constructs discontinuous constituents by allowing the parser to reduce pairs of items that are not adjacent in the stack. In practice, Coavoux and Crabbé (2017a) split the usual stack of shift-reduce parsers into two data structures (a stack and a double-ended queue), in order to give the parser access to two focus items: the respective tops of the stack and the dequeue, that may or may not be adjacent. A dedicated action, GAP, pushes the top of the stack onto the bottom of the queue to make the next item in the stack available for a reduction.

The split stack associated with the GAP action can be interpreted as a linear-access memory: it is possible to access the $i$th element in the stack, but it requires $i$ operations.

**Non-local transitions** Non-local transitions generalize standard parsing actions to non-adjacent elements in the parsing configurations. Maier and Lichte (2016) introduced a non-local transition SKIPSHIFT-$i$ which applies SHIFT to the $i$th element in the buffer. Non-local transitions are also widely used in non-projective dependency parsing (Attardi, 2006; Qi and Manning, 2017; Fernández-González and Gómez-Rodríguez, 2018).

The key difference between these systems and ours is that we use an unordered memory. As a result, the semantics of the COMBINE-$s$ action we introduce in Section 2 is independent from a specific position in the stack or the buffer. A system with an action such as SKIPSHIFT-$i$ needs to learn parameters with every possible $i$, and will only learn parameters with the SKIPSHIFT-$i$ actions that are required to derive the training set. In contrast, we use the same parameters to score each possible COMBINE-$s$ action.

6 Conclusion

We have presented a novel transition system that dispenses with the use of a stack, i.e. a memory with linear sequential access. Instead, the memory of the parser is represented by an unordered data structure with random-access: a set. We have designed a dynamic oracle for the resulting system and shown their empirical potential with state-of-the-art results on discontinuous constituency parsing of one English and two German treebanks. Finally, we plan to adapt our system to non-projective dependency parsing and semantic graph parsing.

Acknowledgments

We thank Caio Corro, Giorgio Satta, Marco Damonte, as well as NAACL anonymous reviewers for feedback and suggestions. We gratefully acknowledge the support of Huawei Technologies.

References


A Oracle Correctness

The oracle o leads to the reachable tree with the highest F-score with respect to the gold tree. The F-score of a predicted tree \( \hat{t} \) (represented as a set of instantiated constituents) with respect to a gold tree \( t^* \) is defined as:

\[
\begin{align*}
\text{precision}(\hat{t}, t^*) &= \frac{|\hat{t} \cap t^*|}{|\hat{t}|}, \\
\text{recall}(\hat{t}, t^*) &= \frac{|\hat{t} \cap t^*|}{|t^*|}, \\
F_1(\hat{t}, t^*) &= \frac{2pr}{p+r}. 
\end{align*}
\]

By definition, \( o_{\text{odd}} \) is optimal for precision because it constructs a constituent only if it is gold, and optimal for recall because it will construct a gold constituent if it is possible to do so.

Moreover, \( o_{\text{even}} \) is optimal for recall because any gold constituent reachable from \( c \) will still be reachable after any transition in \( o_{\text{even}}(c, t^*) \).

Assuming a configuration \( c = (S, s_f, i, C) \) and \( \text{next}(c, t^*) = s_g \), we consider separately the SHIFT case and the COMBINE-s case:

- **SHIFT case** (\( \max(s_g) > \max(s_f) \)): constituents \( (X, s) \) reachable from \( c \) and not reachable from \( \text{SHIFT}(c) \) satisfy \( \max(s) = i \).
  
  If a gold constituent satisfies this property, we have \( s \preceq s_g \), which contradicts the assumption that \( s_g = \text{next}(c, t^*) \) (see definition of oracle in Section 2.2.2).

- **COMBINE-s case**: Let \( (X, s') \) be a reachable gold constituent. Since it is compatible with \( s_g \), there are three possible cases:
  
  - if \( (X, s') \) is an ascendant of \( s_g \), then \( (s \cup s_f) \subseteq s_g \subset s' \), therefore \( (X, s') \) is still reachable from \( \text{COMBINE-s}(c) \).
  
  - if \( (X, s') \) is a descendant of \( s_g \) then \( s' \preceq s_g \), which contradicts the definition of \( s_g \).
  
  - if \( s' \) and \( s_g \) are completely disjoint, we have \( s' \cap s = s' \cap s_f = \emptyset \), therefore \( s' \cap (s \cup s_f) = \emptyset \), and \( s' \) is still reachable from \( \text{COMBINE-s}(c) \).

Finally, since \( o_{\text{even}} \) does not construct new constituents (it is the role of labelling actions), it is optimal for precision.

B Hyperparameters

The list of hyperparameters is presented in Table 5.

- We use learning rate warm-up (linear increase from 0 to \( t_{1000} \) during the first 1000 steps).
- Before the \( t^{th} \) update, we add Gaussian noise to the gradient of every parameter with mean 0 and variance \( \frac{0.01}{(1 + t)^{0.55}} \) (Neelakantan et al., 2015).
- All experiments use greedy search decoding (we did not experiment with beam search).
- Before each training step, we replace a word embedding by an ‘UNK’ pseudo-word embedding with probability 0.3. We only do this replacement for the least frequent word-types (\( \frac{2}{3} \) least frequent word-types). The ‘UNK’ embedding is then used to represent unknown words.
- We apply dropout at the input of the tagger and the input of action scorers: each single prediction has its own dropout mask.

C Detailed Results
Architecture hyperparameters

| Dimension of word embeddings | 32 |
| Dimension of character embeddings | 100 |
| Dimension of character bi-LSTM state | 50 for each direction |
| Dimension of sentence-level bi-LSTM | 200 for each direction |
| Dimension of hidden layers for the action scorer | 200 |
| Activation functions | $tanh$ for all hidden layers |

Optimization hyperparameters

| Initial learning rate | $l_0 = 0.01$ |
| Learning rate decay | $l_t = \frac{l_0}{1 + t \cdot 10^{-7}}$ for step number $t$ |
| Dropout for tagger input | 0.5 |
| Dropout for parser input | 0.2 |
| Training epochs | 100 |
| Batch size | 1 sentence |
| Optimization algorithm | Averaged SGD (Polyak and Juditsky, 1992; Bottou, 2010) |
| Word and character embedding initialization | $U([-0.1, 0.1])$ |
| Other parameters initialization (including LSTMs) | Xavier (Glorot and Bengio, 2010) |
| Gradient clipping (norm) | 100 |
| Dynamic oracle $p$ | 0.15 |

Table 5: Hyperparameters of the model.

<table>
<thead>
<tr>
<th>Parser</th>
<th>Setting</th>
<th>Tiger</th>
<th>DPTB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This work</strong></td>
<td>Python, neural, greedy, CPU</td>
<td>978</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>910</td>
<td>38</td>
</tr>
<tr>
<td>MTG (Coavoux et al., 2019)</td>
<td>C++, neural, greedy, CPU</td>
<td>1934</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1887</td>
<td>80</td>
</tr>
<tr>
<td>MTG (Coavoux and Crabbé, 2017a)</td>
<td>C++, perceptron, beam=4, CPU</td>
<td>4700</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>rparse (Maier, 2015)</td>
<td>Java, perceptron, beam=8, CPU</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>640</td>
<td></td>
</tr>
<tr>
<td>Corro et al. (2017)</td>
<td>C++, neural, CPU</td>
<td>7.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Parsing times on development sets in tokens per second (tok/s) and sentences per second (sent/s). The parsing times are presented as reported by authors, they are not comparable across parsers (since the experiments were run on different hardware). Our parser is run on a single core of an Intel i7 CPU.

<table>
<thead>
<tr>
<th>Development sets</th>
<th><strong>All const.</strong></th>
<th><strong>Disc. const.</strong></th>
<th>POS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>English (DPTB)</td>
<td>static</td>
<td>91.1</td>
<td>91.1</td>
</tr>
<tr>
<td></td>
<td>dynamic</td>
<td>91.4</td>
<td>91.5</td>
</tr>
<tr>
<td>German (Tiger)</td>
<td>static</td>
<td>87.4</td>
<td>87.8</td>
</tr>
<tr>
<td></td>
<td>dynamic</td>
<td>87.6</td>
<td>88.2</td>
</tr>
<tr>
<td>German (Negra)</td>
<td>static</td>
<td>83.6</td>
<td>83.8</td>
</tr>
<tr>
<td></td>
<td>dynamic</td>
<td>84.0</td>
<td>84.7</td>
</tr>
<tr>
<td>Test sets</td>
<td>F</td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>English (DPTB)</td>
<td>dynamic</td>
<td>90.9</td>
<td>91.3</td>
</tr>
<tr>
<td>German (Tiger)</td>
<td>dynamic</td>
<td>82.5</td>
<td>83.5</td>
</tr>
<tr>
<td>German (Negra)</td>
<td>dynamic</td>
<td>83.2</td>
<td>83.8</td>
</tr>
</tbody>
</table>

Table 7: Detailed results. Overall, the positive effect of the dynamic oracle on Fscore is explained by its effect on precision.