Discontinuous parsing with continuous trees

Wolfgang Maier and **Timm Lichte**
Institute for Language and Information
University of Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
{maierwo,lichte}@phil.hhu.de

Abstract

We introduce a new method for incremental shift-reduce parsing of discontinuous constituency trees, based on the fact that discontinuous trees can be transformed into continuous trees by changing the order of the terminal nodes. It allows for a clean formulation of different oracles, leads to faster parsers and provides better results. Our best system achieves an F_1 of 80.02 on TIGER.

1 Introduction

Certain structures in natural language can be described as *discontinuous*, in the sense that they consist of two or more parts which are not adjacent. In linguistics, such structures are typically considered the result of some kind of movement of an element out of a "base" position. Discontinuous structures occur across many other languages (Huck and Ojeda, 1987). Sentence (1) is a German example, taken from the NeGra treebank.

(1) Darüber muss nachgedacht werden
Thereof must thought-about be
'We have to think about that'

In this sentence, the adverb *Darüber*, modifier of the participle *nachgedacht*, is moved to the front. Treebank annotation generally accounts for such structures: Either, the base position of an element is marked with a trace node which is coindexed with the moved element, as it is done, e.g., in the Penn Treebank; or, all parts of a discontinuous constituent are grouped under a single node, as it is done in the German TIGER and NeGra treebanks. This can be seen in Fig. 1, which shows the treebank annotation of (1). The connection between *Darüber* and its reference participle *nachgedacht* is made by grouping both words under a single VP node.

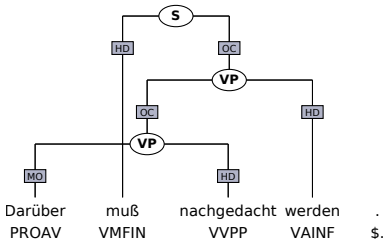

Figure 1: Discontinuous annotation of (1)

Parsing discontinuous constituents is a challenge, since approaches that produce context-free derivations cannot be used. For treebanks in which discontinuities are represented by traces, various approaches have been presented, mostly based on the extension of a CFG parser with a pre-, post- or in-processing step. See, e.g., Johnson (2002), Dienes and Dubey (2003), Levy and Manning (2004), Schmid (2006), and Cai et al. (2011). For the direct parsing of discontinuous constituents, grammar-based techniques have been used, mostly on the basis of Linear Context-Free Rewriting System (LCFRS) (Vijay-Shanker et al., 1987). LCFRS is an extension of Context-Free Grammar in which a single non-terminal can span $k \geq 1$ continuous parts of the input string; i.e., CFG is a special case of LCFRS in which $k = 1$. See, e.g., Kallmeyer and Maier (2013), van Cranenburgh (2012), Angelov and Ljunglöf (2014), Nederhof and Vogler (2014),

Proceedings of DiscoNLP 2016, pages 47–57,
San Diego, California, June 17, 2016.

or Cohen and Gildea (2015). Even with advanced approaches such as the latter, the high parsing complexity with such approaches is a major bottleneck which tends to lead to low parsing speeds.

Another approach consists of creating a reversible conversion of discontinuous constituents to dependencies, and to parse those with an appropriate dependency parser. This very successful approach is taken by Hall and Nivre (2008) and Fernández-González and Martins (2015).

Recently, Versley (2014) and Maier (2015) have exploited a strategy known from non-projective dependency parsing (Nivre, 2009; Nivre et al., 2009): One can convert every non-projective dependency tree into a projective one by reordering its words. Non-projective dependency parsing can therefore be cast as projective dependency parsing with an additional online reordering operation which allows for the input to be processed out-of-order ("swap"). The same holds for the parsing of discontinuous constituency trees. While Versley (2014) adapts the "easy-first" strategy of Goldberg and Elhadad (2010) to work with a swap operation, Maier (2015) extends the shift-reduce approach of Zhu et al. (2013) correspondingly. Note that the idea of processing linear precedence and immediate dominance for discontinuous parsing separately has also been explored in a grammar-based context by Nederhof and Vogler (2014).

In this paper, we build on the work of Maier (2015) and make two contributions. Firstly, we introduce a new parser transition SKIPSHIFT-i which in comparison to the swap operation reduces the amount of decisions required to be taken in order to produce a discontinuous constituent and therefore leads to fewer errors. Secondly, we address the problem that when processing the input terminals out-of-order, the same tree can be mapped to different parser transition sequences. We introduce an algorithm which reorders the terminals of a tree off-line such that the resulting tree is continuous. The reordered terminals are used as a basis for obtaining an oracle that maps the tree to a canonical transition sequence. All new techniques are implemented within *uparse*, the publicly available parser of Maier (2015).[1] An experimental evaluation shows that choosing the appropriate terminal order is crucial to parsing success: We obtain state-of-the-art results for discontinuous shift-reduce constituency parsing, namely 80.02 on the TIGER data set of Hall and Nivre (2008).

The remainder of the article is organized as follows. In Sec. 2, we present the basic architecture for shift-reduce parsing with discontinuous constituents of Maier (2015), on which we build our work. Sec. 3 presents our methodology. In Sec. 4, we present our experiments and discuss the results, and Sec. 5 closes the article.

2 Discontinuous Shift-Reduce Parsing

2.1 Basic parser architecture

We base our work on the shift-reduce parser architecture of Maier (2015), which in turn is based on the architecture of Zhu et al. (2013).

As usual in shift-reduce parsing, a parser *state* represents a (partial) derivation. It is a tuple consisting of a *queue* of incoming pairs of input tokens and POS tags[2] which have not yet been processed, and a stack which holds completed constituents. The *initial* parser state has an empty stack and a queue which holds the input string to be parsed. From a given state, other states can be reached via the following state transitions.

- SHIFT shifts a single element from the queue onto the stack.
- BINL-X, resp. BINR-X build a new X constituent with the first two stack elements as its children and its head coming from its left, resp. right child. The new constituent replaces the first two stack elements.
- UNARY-X builds a new X constituent with the first stack element as its child. The new constituent replaces the first stack element.
- SWAP-i handles discontinuous constituents. It allows for a block of i elements from the stack

[1] https://github.com/wmaier/uparse.

[2] POS tagging and constituency shift-reduce parsing can be done jointly, as demonstrated, e.g., by Mi and Huang (2015). However, note that throughout, we assume POS tagging to be done outside of the parser as in earlier work (Zhu et al., 2013; Maier, 2015).

to be swapped back on the queue, starting with the second stack element.[3]

- FINISH pops the last remaining element from the stack, given that it is labeled with the root label and the queue is empty.

- IDLE can be applied any number of times after FINISH. This compensates for different lengths of analyses (Zhu et al., 2013).

A parser state to which FINISH has been applied is a *final* state. Transitions can only be applied on states which fulfill certain conditions. For instance, SHIFT can only be applied if there are elements left on the queue. The full set of the corresponding conditions is listed in the appendix of Zhang and Clark (2009), and in Maier (2015) (for SWAP).

2.2 Oracles

An *oracle* is used to obtain canonical transition sequences from gold treebank trees. These sequences can then be learned by the parser.

Since we only use transitions that handle unary and binary nodes, incoming trees must be binarized. As in previous work, we use head-outward binarization with binary top and bottom productions, and a single binarization label @X for all X constituents. For details on the binarization, see Maier (2015) and references therein.

For continuous parsing, i.e., when no discontinuous trees have to be handled such as in Zhu et al. (2013), one can use a simple oracle which traverses the tree top-down in postorder. Before the traversal starts, a single IDLE followed by a single FINISH transition are generated. Then, at each binary X node, a corresponding BINL-X / BINR-X is generated; each unary X node leads to a UNARY-X transition. For a terminal, SHIFT is generated. In a last step the transition sequence is reversed.

Discontinuous trees can be handled with the SWAP-i transition, which allows for the input to be processed out-of-order (Maier, 2015). The corresponding oracle traverses a treebank tree bottom up left-to-right. Mirroring the parser, it maintains an incoming list of token/POS-pairs to be processed (initially filled with the sequence of terminals/preterminals of the tree), a stack structure holding subtrees of the treebank tree to be processed (initially empty), and a list holding the result.

While the incoming list is not empty, repeat the following two steps.

1. If the stack is not empty, repeat:

   - If the root of the first subtree on the stack is the only child of its parent node p (labeled X), pop the subtree from the stack and push the subtree the root of which is p, then add an UNARY-X transition to the result.
   - If the first two subtrees on the stack share the same parent node p (labeled X), pop both subtrees from the stack and push the subtree the root of which is p, then add a BINR-X / BINL-X transition, depending on the head side of p.

   If no transitions have been added, go to the next step.

2. If the incoming list is not empty, process the next terminal as follows. Determine the leftmost terminal dominated by the right child of the parent of the top element on the stack. If there is no gap, then this is the first element in our incoming list. We can add a single SHIFT, remove the first element from the incoming list and add it to the stack. If there is a gap, then there are $i \geq 1$ terminals between the head of the list and the terminal to be shifted. We therefore add $i+1$ SHIFT and one SWAP-i transition; then we remove the $i+1$st element from the incoming list and add it to the stack.

Note that in the continuous case, both oracles yield the same result. As an example, consider Fig. 1. At the start, no reduce transitions can be added because the stack is empty. We pass to step 2, add a SHIFT transition to the result and remove the first token (*Darüber*) from the list. Then, we must jump over the gap, i.e., we determine the leftmost terminal dominated by the right child of the parent of the topmost stack element. The parent of the topmost stack element is VP, and its rightmost child *nachgedacht*, resp. VVPP. We therefore

[3] This operation is called COMPOUNDSWAP$_i$ in Maier (2015); we do not use single SWAP here.

add two SHIFTs and one SWAP-1. This then allows for the addition of a BINR-VP. In the following, we have to jump over *muß* again. The parent of the topmost stack element is (the upper) VP and its rightmost child is *werden*. Therefore, we add again two SHIFTs and one SWAP-1 followed by BINR-VP. Last, we SHIFT the remaining *muss* and add BINR-S.

2.3 Structured prediction

For structured prediction, each parser state is assigned a score. The score of the start state is 0, and the score of the $i+1$th state is the sum of the score of the ith state and the dot product of a local feature vector obtained through a feature function ϕ and a global weight vector θ. We train θ with the averaged Perceptron using max violation (Huang et al., 2012). Decoding is done with beam search. The beam of size n is initialized with the start state. Then repeatedly, a candidate list is filled with all states which can be built using transitions on states on the beam. The highest scoring n of them are put back on the beam. Parsing is finished if the highest scoring state on the beam is a final state.

2.4 Features

The feature function ϕ works by applying templates to parser states. The templates describe particular configurations of stack and queue. As does Maier (2015), we use the feature template set from Zhang and Clark (2009) as a baseline, and furthermore experiment with the extended features of Zhu et al. (2013). We also use the features for discontinuities from Maier (2015). For the full template list, consult Appendix A.

3 Methodology

We now present the main contributions of this paper, namely, the tree reordering algorithm, the new parser transition, and the new oracles.

3.1 Discontinuous Trees

First, we define the structures which are manipulated by the tree reordering algorithm. A *discontinuous tree* is a directed acyclic graph $T = (V, E, r)$ where V is a set of nodes with $r \in V$ the *root* node, and $E : V \times V$ a set of edges with E^* the reflexive transitive closure of E. For all $v \in V \setminus \{r\}$, there is exactly one $(u, v) \in E$ with $u \in V$; u is thereby called the *parent* of v and v a *child* of u. Nodes with no children are called *terminals*. We let $V_T = \{v \in V \mid v \text{ is a terminal}\}$; all $v_t \in V_T$ are uniquely numbered from 1 to $|V_T|$ by a function ind (Maier and Lichte, 2011). The *yield* of a node $v \in V$ is the set of all terminals it dominates, i.e., for all $v \in V$, $yield(v) = \{u \in V_T \mid (v, u) \in E^*\}$. V is equipped with an order $\prec$, which orders the nodes according to the lowest index of the terminals they dominate. I.e., $u \prec v$ iff $\min(\{ind(u') \mid u' \in yield(u)\}) < \min(\{ind(v') \mid v' \in yield(v)\})$ where $u, v \in V$. We write $c(v)$ for the ordered list of the children of a given node v; for the ith child of v, we write $c(v)[i]$.

Any node $v \in V$ is *discontinuous* (as opposed to *continuous*) if there are $v_1, v_2 \in yield(v)$ such that $ind(v_1) + 1 < ind(v_2)$ and $ind(v_1) + 1 \notin \{ind(v') \mid v' \in yield(v)\}$. If $\{i \mid v_1 < i < v_2\} \cap ind(v) = \emptyset$, then the tuple (v_1, v_2) is called a *gap* of size $ind(v_2) - ind(v_1) - 1$; v_1 and v_2 are called its left and right *border*. T is discontinuous if V contains discontinuous nodes. A continuous node which has discontinuous children is called a *gap creator*.

3.2 Continuous Tree Reordering

As already observed in previous literature (Nivre, 2009; Versley, 2014; Maier, 2015), a discontinuous tree can be made continuous by changing the order of the terminals. The new terminal order must be such that all gaps in yields are eliminated, i.e., it must ensure that discontinuous parts are joined.

In other words, given a tree (V, E, r) we want to produce a function ind_σ to replace ind. ind_σ must be such that all $v \in V$ are continuous. In general, more than one such function will exists for a tree. As an example, consider again the tree in Fig. 1. Two possible permutations of the terminals which eliminate the VP gap would be: *Darüber nachgedacht werden muß* (i.e., ind_σ(Darüber) = 1, ind_σ(nachgedacht) = 2, ind_σ(werden) = 3, ind_σ(muß) = 4) and *muß Darüber nachgedacht werden* (i.e., ind_σ(muß) = 1, ind_σ(Darüber) = 2, ind_σ(nachgedacht) = 3, ind_σ(werden) = 4). Note that the first order is the complementizer-free order of embedded sentences. The second order is also an acceptable German sentence, namely, the question

form of the original sentence.

Given a binarized tree $T = (V, E, r)$, we can obtain different variants of ind_σ systematically with a recursive top-down procedure *reorder*, to be called on r. *reorder* yields a bijection σ of the set $\{ind(v_t) \mid v_t \in yield(r)\}$ to itself, and we define ind_σ such that for all $v_t \in yield(r)$, $ind_\sigma(v_t) = \sigma(ind(v_t))$. Note that we use tuple notation with angled brackets for the output of the bijection; $\oplus$ denotes tuple concatenation

When called on a node $v \in V$, *reorder* distinguishes three cases. (1) If v is a terminal, $reorder(v)$ simply yields $[ind(v)]$. (2) If v is an unary node, $reorder(v)$ yields $reorder(c(v)[0])$. (3) If v is a binary node, then there are two different possibilities. *reorder* can yield $reorder(c(v)[0]) \oplus reorder(c(v)[1])$ or $reorder(c(v)[1]) \oplus reorder(c(v)[0])$. The former is called *left* reordering, since all terminals of the left child are placed in front of all terminals of the right child. Correspondingly, the latter is called the *right* reordering. Not all nodes have to be reordered in the same way; we therefore explore the following reordering selections.

- LEFT: always select left reordering.
- RIGHT: always select right reordering. This also reorders continuous nodes.
- RIGHTD: select right reordering if node is a gap creator, otherwise do not reorder.
- DISTi: select right reordering if node is a gap creator and the left or right child have a gap of size $\geq i$, otherwise do not reorder.
- LABEL: specify a reordering direction for particular labels, and a default reordering for all others.

As an example, let us look at the result of RIGHTD when applied on the tree in Fig. 1. The original set of indices of the root node is $\{1, 2, 3, 4\}$. Since S is binary and a gap creator (S is continuous, but its VP child is discontinuous), we select the right reordering, i.e., $reorder$(S) is $reorder$(muß) $\oplus$ $reorder$(VP). $reorder$(muß) is [2]. The VP is not a gap creator, i.e., we apply the left reordering and obtain $reorder$(VP) $\oplus$ $reorder$(werden). $reorder$(werden) is [4]. The lower VP is also no gap creator, therefore we apply again the left reordering and obtain $reorder$(Darüber) $\oplus$ $reorder$(nachgedacht). $reorder$(Darüber) is [1], and $reorder$(nachgedacht) is [3], i.e., the result for the lower VP is $[1, 3]$. The result for the upper VP is $[1, 3, 4]$, and the result for S is $[2, 1, 3, 4]$. In other words, we obtain ind_σ(Darüber) $= 2$, ind_σ(muss) $= 1$, ind_σ(nachgedacht) $= 3$, and ind_σ(werden) $= 4$, the above mentioned question ordering of the sentence.

3.3 From Swap to Skip-Shift

We introduce a new transition SKIPSHIFT-i, which shifts the ith element from the queue (counting from 0). In other words, it allows the parser to directly skip elements on the queue while shifting. SKIPSHIFT-i underlies the same restrictions as the "regular" SHIFT transition (Zhang and Clark, 2009): The queue must not be empty and the state must not be a final state. Furthermore, if last transition was of type BINR- and led to an intermediate constituent, SKIPSHIFT-i is disallowed.

The new transition reduces the amount of transitions needed to parse a discontinuity. As an example, consider the transition sequence which would be extracted from the tree in Fig. 1 with SHIFT and SWAP, namely SHIFT, SHIFT, SHIFT, SWAP, BINR-VP, SHIFT, SHIFT, SWAP, BINR-VP, SHIFT, BINR-S. Note that the word *muss* is shifted two times and swapped back on the queue before it is finally used in a reduce transition the third time it is shifted. With SKIPSHIFT-i, only 7 instead of 11 decisions are required: SKIPSHIFT-0, SKIPSHIFT-1, BINL-VP, SKIPSHIFT-1, BINL-VP, SKIPSHIFT-0, BINR-S.

3.4 From reordering to oracles

In the case of discontinuities, the bottom-up oracle from Maier (2015) determines the next terminal to be shifted by skipping over the discontinuity through a traversal of the relevant part of the input tree, namely, the path from the left to right border of a gap.

We modify the old oracle such that the order of operations is determined by the terminal reordering obtained with the *reorder* procedure described in Sec. 3.2. If the incoming terminal list is ordered by

ind_σ, the gap borders in the original ordering are joined together. Therefore, no tree walk is necessary to determine the next terminal to be shifted (i.e., the right border of the gap). It is always the first one in the reordered list. We must, however, determine how many terminals have to be shifted and swapped, resp. skipped, relative to the original order, i.e., the size i of the gap that is skipped. For this, we determine the index of the first terminal in the list of terminals to be processed, and then count the number of terminals in the list that have a lower index; i is set to this number minus one. Once determined, i can be used with either SHIFT and SWAP, or with SKIPSHIFT. With the former, we generate $i + 1$ SHIFT followed by one SWAP-i transition. With the latter, we just generate a single SKIPSHIFT-i transition.

3.5 More features

The new SKIPSHIFT-i operation can access any element in the input queue, not just the first one. In order to make the corresponding information available to the parser, we introduce a new feature set which consist of all token/POS pairs remaining on the queue ("full queue features").

Furthermore, we adapt *swap importance weighting* from Maier (2015). During training, all updates that concern a swap transition are counted twice. We do the same for SKIPSHIFT-i.

4 Experiments

We implement the tree transformations with the corresponding oracles, as well as the SKIPSHIFT-i operation within `uparse`, the publicly available implementation of the parser of Maier (2015).[4]

4.1 Data and Setup

In order to facilitate a comparison, we use the same data as Maier (2015), we use the TIGER treebank release 2.2 with the splits of Farkas and Schmid (2012) and Hall and Nivre (2008). In the former, the first half of the last 10,000 sentences are used for development and the second half for testing, and the rest is left for training. In the latter, the treebank is split into ten parts, such that sentence i is put into part $i \; mod \; 10$. The first of those parts is used for testing, the concatenation of the rest for training. As usual, we attach the material which is not included in the annotation (mainly punctuation) to the tree itself.

We run the training for 20 iterations using beam size 4. Other parameters are indicated later. The results are reported as labeled bracket scores, as obtained with the evaluation module of `discodop`.[5] We use the `proper.prm` file of the `discodop` distribution, i.e., root nodes are not included, but punctuation is included in the evaluation.

4.2 Results

All experimental results are listed in Tab. 1 (evaluation on all constituents) and Tab. 2 (evaluation only on discontinuous constituents).

As a baseline, we run a single experiment with the SWAP-i transition. Then, with different settings, we run experiments with the SKIPSHIFT-i transition, using the LEFT, RIGHTD, RIGHT, LABEL, and the DISTi reorderings, choosing 2, 4, and 8 as distance for the latter. For the label-based reordering, we employ the left reordering for all NP and PP nodes, and RIGHTD for all other nodes. NPs and PPs are very similar in the TIGER annotation; the only difference between both is the presence of a preposition in PPs, resp. the absence of it in NPs (Maier et al., 2014). Both are often deeply embedded; we therefore presume that recognizing the entire phrase *before* recognizing the material in the gap is more promising.

The results for SWAP-i confirm the results from Maier (2015). With the SKIPSHIFT-i operation, the parser performs consistently better than with SWAP-i. The best result is obtained with DIST2, an F_1 gain of 2.2 over the swap baseline. Unsurprisingly, RIGHT does not perform well due to the fact that it also aggressively reorders nodes which are continuous; it will not be included in the remaining experiments. When evaluating discontinuous constituents only, DIST8 is the most successful setting. This can be explained by the fact that preferring the right reordering in nodes with children that have large gaps means that the i in SKIPSHIFT-i transitions can be maintained lower than with the left reordering. Since SKIPSHIFT-i with a lower i are seen more frequently in training, results improve. LABEL is not that successful, achieving only a slightly higher precision on discontinuous constituents.

[4] `https://github.com/wmaier/uparse`.

[5] `https://github.com/andreasvc/discodop`.

	Rec.	Prec.	F_1	Exact
SWAP-i	72.80	74.67	73.72	36.27
Baseline				
LEFT	73.94	75.54	74.73	37.37
RIGHTD	75.15	76.77	75.95	37.19
RIGHT	25.61	22.53	23.97	10.68
LABEL	75.14	76.71	75.92	37.28
DIST2	75.13	76.81	75.96	37.19
DIST4	74.98	76.55	75.76	37.19
DIST8	75.18	76.64	75.88	37.56
+ *extended features*				
LEFT	74.43	75.87	75.14	37.80
RIGHTD	75.58	77.07	76.32	37.86
LABEL	75.24	76.61	75.92	37.41
DIST2	75.70	**77.25**	**76.46**	37.82
DIST4	75.60	77.07	76.33	37.84
DIST8	75.18	76.61	75.89	37.83
+ *extended and disco features*				
LEFT	74.77	76.10	75.43	37.84
RIGHTD	**75.73**	77.14	76.43	37.13
LABEL	75.65	77.07	76.35	37.38
DIST2	75.57	76.95	76.25	37.69
DIST4	75.63	77.08	76.35	37.52
DIST8	75.39	76.77	76.08	37.72
+ *extended and full queue features*				
LEFT	74.44	75.96	75.19	**38.54**
RIGHTD	75.42	76.96	76.18	38.20
LABEL	75.30	76.91	76.10	37.90
DIST2	75.65	77.17	76.40	38.01
DIST4	75.43	76.83	76.12	38.22
DIST8	75.20	76.54	75.87	37.95
+ *extended features and imp. weighting*				
LEFT	74.45	75.82	75.13	37.53
RIGHTD	75.44	76.77	76.10	37.48
LABEL	75.42	76.84	76.12	37.70
DIST2	75.51	76.89	76.19	37.37
DIST4	75.18	76.55	75.86	37.68
DIST8	75.00	76.36	75.67	37.12
+ *extended, disco and full queue feature* + *importance weighting*				
LEFT	74.87	76.23	75.54	37.61
RIGHTD	75.28	76.63	75.95	37.03
LABEL	75.30	76.76	76.02	36.94
DIST2	75.43	76.95	76.19	37.54
DIST4	75.33	76.65	75.98	37.05
DIST8	75.16	76.64	75.89	37.28

Table 1: Results (all constituents)

	Rec.	Prec.	F_1	Exact
SWAP-i	10.29	23.64	14.34	8.03
Baseline				
LEFT	14.31	18.64	16.19	12.33
RIGHTD	10.90	25.27	15.24	8.74
RIGHT	1.73	0.18	0.33	0.51
LABEL	10.58	25.88	15.02	8.75
DIST2	11.82	28.15	16.64	9.73
DIST4	11.63	25.40	15.96	10.10
DIST8	13.53	27.11	18.05	11.29
+ *extended features*				
LEFT	13.54	21.38	16.58	12.20
RIGHTD	10.25	31.00	15.41	9.04
LABEL	10.19	27.92	14.94	8.99
DIST2	11.02	31.32	16.31	9.99
DIST4	11.22	29.16	16.20	10.25
DIST8	11.78	26.00	16.21	10.56
+ *extended and disco features*				
LEFT	**14.90**	29.53	19.80	14.17
RIGHTD	8.43	42.13	14.05	8.14
LABEL	8.87	45.91	14.86	8.57
DIST2	9.03	41.49	14.83	9.43
DIST4	10.14	42.86	16.40	9.62
DIST8	11.42	37.96	17.55	11.06
+ *extended and full queue features*				
LEFT	14.30	24.86	18.16	12.98
RIGHTD	10.96	30.69	16.15	9.79
LABEL	11.12	29.02	16.08	9.57
DIST2	11.58	31.93	17.00	9.91
DIST4	11.03	27.74	15.78	9.81
DIST8	12.06	27.60	16.79	10.42
+ *extended features and imp. weighting*				
LEFT	13.39	24.24	17.25	12.36
RIGHTD	9.15	31.98	14.23	7.79
LABEL	9.47	33.83	14.80	8.32
DIST2	10.00	33.59	15.41	9.12
DIST4	9.70	32.50	14.94	9.41
DIST8	10.54	29.14	15.48	9.78
+ *extended, disco and full queue feature* + *importance weighting*				
LEFT	14.31	35.27	**20.36**	**13.80**
RIGHTD	8.91	**45.93**	14.92	8.37
LABEL	8.39	43.10	14.04	8.44
DIST2	8.45	42.62	14.11	8.67
DIST4	8.91	41.11	14.65	8.72
DIST8	10.95	41.84	17.35	10.55

Table 2: Results (discontinuous constituents)

	V	M	**here**	vC	H&N	F&M
F_1	74.23	79.52	80.02	79.00	79.93	**85.53**
E	37.32	44.32	45.11	41.33	37.78	**51.21**

Table 3: Results for sentence length ≤ 40 on H&N data

The fact that we need less operations in total in order to build a tree (in comparison to SWAP-i) is reflected in reduced parsing times. With SWAP-i, we need 63 seconds to parse the entire test set (79.5 sent./sec.), with SKIPSHIFT-i and LEFT, we only need 49 seconds (101.8 sent./sec.).

When adding the extended features from Zhu et al. (2013), the trend seen in Maier (2015) is repeated: Looking deeper into the structures on the stack leads to a higher performance when looking at all constituents. On discontinuous constituents, we obtain a improved precision, but a slightly worse recall. The features for discontinuities have also the same effect as in Maier (2015), no improvement is achieved on all constituents. The precision on discontinuous constituents is, however, much higher while the corresponding recall drops sharply, indicating data sparseness. Adding the full queue features to the extended features only has a very small effect.

Also, adding importance weighting alone is not very successful. However, when we combine the extended, the discontinuous and the full queue features with importance weighting, we achieve the best result on discontinuous constituents; surprisingly this happens when using the LEFT reordering.

Last, for comparison, we run experiments on the Hall and Nivre (2008) (H&N) data set. We run our own parser (LEFT; extended, discontinuous, and full queue features; importance weighting); and also `discodop` (van Cranenburgh and Bod, 2013) (vC) (default settings, using gold POS tags). Tab. 3 shows the corresponding results along with those of Versley (2014) (V), Maier (2015) (M), H&N and Fernández-González and Martins (2015) (F&M) (taken from Maier (2015)). Note that we do improve on M, but still lie much behind F&M.

4.3 Discussion

What about the big picture? In spite of an improvement on Maier (2015), the scores on the discontinuous constituents remain very low. A manual analysis of the parsing results leads us to the conclusion that the strongest point of discontinuous shift-reduce parsing, namely the locality of the search which leads to its speed, is also its biggest weakness. Certain structures can simply not be recognized almost "by definition". For instance, in order to correctly recognize an NP with an extraposed modifier, the reduction of the full NP must be delayed until the complete modifier has been recognized. With the current parsing model, in some situations, there is just no way of knowing if a delay is necessary, since the modifier can be still out of reach for the feature function when the first part of the full NP has already been recognized. Due to beam search, once the NP is reduced, we cannot backtrack, i.e., the modifier cannot be attached later.

One way of addressing this problem could be the use of exact search, such as in Thang et al. (2015). However, there is another perspective. An important finding of the experiments is that recognizing material in gaps *before* the discontinuous constituent itself (RIGHTD) leads to high precision and low recall on discontinuous constituents, while recognizing the discontinuous constituent first (LEFT) leads to more errors, but also catches more cases (lower precision, higher recall). The reason for this is that in the former case, one decides too late and in the latter case too early if a partially recognized constituent is part of a discontinuous structure. We conjecture that what makes the parsers of Fernández-González and Martins (2015) and van Cranenburgh and Bod (2013) successful are their mechanisms of handling this issue: The former joins the recognition of a terminal with the decision of what part of a (potentially discontinuous) constituent it belongs to. The latter gets structure-global context by building the final tree as a combination of discontinuous base structures. This makes it particularly more successful on discontinuous structures, achieving Prec./Rec./F_1 of 33.77/50.29/40.41 on them (compared to our result of 19.36/39.71/26.03) (unfortunately, Fernández-González and Martins (2015) have not reported results on discontinuous constituents alone). In future work, we will explore possibilities of integrating such a mechanism in a shift-reduce approach.

We have seen that choosing the transition order well, i.e., picking the right oracle, is crucial for parsing success. We therefore want to explore how the

order of transitions affects parsing results in the continuous case. A particular transition order could be forced via a tree transformation such as *right-corner transform* (Schuler et al., 2010). Concretely, right-corner transform would give preference to easier transition orders in which we reduce as soon as possible, i.e., long sequences of SHIFTs would be avoided.

Last, it should be noted that the application of SKIPSHIFT-i is not limited to discontinuous constituency parsing. We want to apply our method to non-projective dependency parsing, where SKIPSHIFT-i could be used instead of swap-eager/lazy transitions (Nivre et al., 2009) in a parsing framework such as the one of Zhang and Nivre (2011). This would also allow for an "intersection" between our work and then one of Fernández-González and Martins (2015).

5 Conclusion

We have presented a new tree reordering method which makes discontinuous constituency trees continuous. The reordering method can be used to obtain oracles for discontinuous shift-reduce parsing. In conjunction with a new parser transition, we have achieved state-of-the-art results for discontinuous shift-reduce constituency parsing.

Appendix A. Feature Templates

Our parser uses feature templates from previous work. For the sake of completeness, we list them here. s_i and q_i stand for the ith item on stack and queue, w is the head word, t the head tag and c the constituent label (w, t and c are identical on preterminal level). l and r (ll and rr) are the left and right children (grand-children) of the corresponding element on the stack; u deals with unary constituents.

The following baseline features have been presented by Zhang and Clark (2009).

unigrams

s_0tc, s_0wc, s_1tc, s_1wc, s_2tc, s_2wc, s_3tc, s_3wc,
q_0wt, q_1wt, q_2wt, q_3wt,
s_0lwc, s_0rwc, s_0uwc, s_1lwc, s_1rwc, s_1uwc

bigrams

s_0ws_1w, s_0ws_1c, s_0cs_1w, s_0cs_1c, s_0wq_0w, s_0wq_0t,
s_0cq_0w, s_0cq_0t, s_1wq_0w, s_1wq_0t, s_1cq_0w, s_1cq_0t,
q_0wq_1w, q_0wq_1t, q_0tq_1w, q_0tq_1t

trigrams

$s_0cs_1cs_2w$, $s_0cs_1cs_2c$, $s_0cs_1cq_0w$, $s_0cs_1cq_0t$,
$s_0cs_1wq_0w$, $s_0cs_1wq_0t$, $s_0ws_1cs_2c$, $s_0ws_1cq_0t$

The *extended* features have been introduced by Zhu et al. (2013).

extended

s_0llwc, s_0lrwc, s_0luwc, s_0rlwc, s_0rrwc,
s_0ruwc, s_0ulwc, s_0urwc, s_0uuwc, s_1llwc,
s_1lrwc, s_1luwc, s_1rlwc, s_1rrwc, s_1ruwc

The following *disco* features stem from Maier (2015). As explained there, in the following templates, x denotes the *gap type* of a stack element. It can be "none" (tree on stack is fully continuous), "pass" (there is a gap at the root), and "gap" (the root of this tree fills a gap, i.e., its children have gaps, but the root does not). y stands for the sum of all gap lengths.

unigrams

s_0xwc, s_1xwc, s_2xwc, s_3xwc,
s_0xtc, s_1xwc, s_2xtc, s_3xwc,
s_0xy, s_1xy, s_2xy, s_3xy

bigrams

s_0xs_1c, s_0xs_1w, s_0xs_1x, s_0ws_1x, s_0cs_1x,
s_0xs_2c, s_0xs_2w, s_0xs_2x, s_0ws_2x, s_0cs_2x,
s_0ys_1y, s_0ys_2y, s_0xq_0t, s_0xq_0w

Acknowledgments

We would like to thank Omri Abend for discussions. Thanks also to the three anonymous reviewers for valuable comments and suggestions. This work was partially funded by Deutsche Forschungsgemeinschaft (DFG).

References

Krasimir Angelov and Peter Ljunglöf. 2014. Fast statistical parsing with parallel multiple context-free grammars. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 368–376, Gothenburg, Sweden.

Shu Cai, David Chiang, and Yoav Goldberg. 2011. Language-independent parsing with empty elements. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 212–216, Portland, OR.

Shay B. Cohen and Daniel Gildea. 2015. Parsing linear-context free rewriting systems with fast matrix multiplication. *CoRR*, abs/1504.08342.

Péter Dienes and Amit Dubey. 2003. Antecedent recovery: Experiments with a trace tagger. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, pages 33–40, Sapporo, Japan.

Richard Farkas and Helmut Schmid. 2012. Forest reranking through subtree ranking. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1038–1047, Jeju Island, Korea, July. Association for Computational Linguistics.

Daniel Fernández-González and André F. T. Martins. 2015. Parsing as reduction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and Teh 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, Beijing, China.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 742–750, Los Angeles, CA.

Johan Hall and Joakim Nivre. 2008. Parsing discontinuous phrase structure with grammatical functions. In Bengt Nordström and Aarne Ranta, editors, *Advances in Natural Language Processing*, volume 5221 of *Lecture Notes in Computer Science*, pages 169–180. Springer, Gothenburg, Sweden.

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.

Geoffrey Huck and Almerindo Ojeda, editors. 1987. *Discontinuous constituency*. Academic Press, New York.

Mark Johnson. 2002. A simple pattern-matching algorithm for recovering empty nodes and their antecedents. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 136–143, Philadelphia, PA.

Laura Kallmeyer and Wolfgang Maier. 2013. Data-driven parsing using probabilistic linear context-free rewriting systems. *Computational Linguistics*, 39(1):87–119.

Roger Levy and Christopher Manning. 2004. Deep dependencies from context-free statistical parsers: Correcting the surface dependency approximation. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 327–334, Barcelona, Spain.

Wolfgang Maier and Timm Lichte. 2011. Characterizing discontinuity in constituent treebanks. In *Formal Grammar. 14th International Conference, FG 2009. Bordeaux, France, July 25-26, 2009. Revised Selected Papers*, volume 5591 of *LNCS/LNAI*, pages 167–182, Berlin, Heidelberg, New York. Springer-Verlag.

Wolfgang Maier, Miriam Kaeshammer, Peter Baumann, and Sandra Kübler. 2014. Discosuite - A parser test suite for German discontinuous structures. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland. European Language Resources Association (ELRA).

Wolfgang Maier. 2015. Discontinuous incremental shift-reduce parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1202–1212, Beijing, China, July. Association for Computational Linguistics.

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, May–June. Association for Computational Linguistics.

Mark-Jan Nederhof and Heiko Vogler. 2014. Hybrid grammars for discontinuous parsing. In *Proceedings of COLING 2014, the 25th International Conference on Computation Linguistics: Technical Papers*, pages 1370–1381, Dublin, Ireland.

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.

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, Singapore.

Helmut Schmid. 2006. Trace prediction and recovery with unlexicalized PCFGs and slash features. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 177–184, Sydney, Australia.

William Schuler, Samir AbdelRahman, Tim Miller, and Lane Schwartz. 2010. Broad-coverage parsing using

human-like memory constraints. *Computational Linguistics*, 36(1):1–30.

Le Quang Thang, Hiroshi Noji, and Yusuke Miyao. 2015. Optimal shift-reduce constituent parsing with structured perceptron. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1534–1544, Beijing, China, July. Association for Computational Linguistics.

Andreas van Cranenburgh and Rens Bod. 2013. Discontinuous parsing with an efficient and accurate DOP model. In *Proceedings of The 13th International Conference on Parsing Technologies*, Nara, Japan.

Andreas van Cranenburgh. 2012. Efficient parsing with linear context-free rewriting systems. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 460–470, Avignon, France.

Yannick Versley. 2014. Experiments with easy-first nonprojective constituent parsing. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, pages 39–53, Dublin, Ireland.

K. Vijay-Shanker, David Weir, and Aravind K. Joshi. 1987. Characterising structural descriptions used by various formalisms. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 104–111, Stanford, CA.

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, Paris, France.

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.

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 (Volume 1: Long Papers)*, pages 434–443, Sofia, Bulgaria.