# Discontinuous Constituent Parsing as Sequence Labeling


## Abstract

This paper reduces discontinuous parsing to sequence labeling. It first shows that existing reductions for constituent parsing as labeling do not support discontinuities. Second, it fills this gap and proposes to encode tree discontinuities as nearly ordered permutations of the input sequence. Third, it studies whether such discontinuous representations are learnable. The experiments show that despite the architectural simplicity, under the right representation, the models are fast and accurate.


## 1 Introduction

Discontinuous constituent parsing studies how to generate phrase-structure trees of sentences coming from non-configurational languages, where non-consecutive tokens can be part of the same grammatical function (e.g. nonconsecutive terms belonging to the same verb phrase). Figure 1 shows a German sentence exhibiting this phenomenon. Discontinuities happen in languages that exhibit free word order such as German or Guugu Yimidhirr, but also in those with high rigidity, e.g. English, whose grammar allows certain discontinuous expressions, such as wh-movement or extraposition. This makes discontinuous parsing a core computational linguistics problem that affects a wide spectrum of languages.

There are different paradigms for discontinuous phrase-structure parsing, such as chart-based parsers, transitionbased algorithms or reductions to a problem of a different nature, such as dependency parsing. However, many of these approaches come either at a high complexity or low

speed, while others give up significant performance to achieve an acceptable latency.

Related to these research aspects, this work explores the feasibility of discontinuous parsing under the sequence labeling paradigm, inspired by work on fast and simple continuous constituent parsing. We will focus on tackling the limitations of their encoding functions when it comes to analyzing discontinuous structures, and include an empirical comparison against existing parsers.

Contribution (i) The first contribution is theoretical: to reduce constituent parsing of free word order languages to a sequence labeling problem. This is done by encoding the order of the sentence as (nearly ordered) permutations. We present various ways of doing so, which can be naturally combined with the labels produced by existing reductions for continuous constituent parsing. (ii) The second contribution is a practical one: to show how these representations can be learned by neural transducers. We also shed light on whether general-purpose architectures for NLP tasks can effectively parse free word order languages, and be used as an alternative to adhoc algorithms and architectures for discontinuous constituent parsing.

## 2 Related work

Discontinuous phrase-structure trees can be derived by expressive formalisms such as Multiple Context Free Grammmars (MCFGs) or Linear Context-Free Rewriting Systems (LCFRS). MCFGs and LCFRS are essentially an extension of Context-Free Grammars (CFGs) such that non-terminals can link to non-consecutive spans. Traditionally, chart-based parsers relying on this paradigm

commonly suffer from high complexity. Let k be the block degree, i.e. the number of nonconsecutive spans than can be attached to a single non-terminal; the complexity of applying CYK (after binarizing the grammar) would be O(n3k), which can be improved to O(n2k+2) if the parser is restricted to well-nested LCFRS, and discusses how for a standard discontinuous treebank, k 3 (in contrast to k = 1 in CFGs). Recently, presents a chart-based parser for k = 2 that can run in O(n3), which is equivalent to the running time of a continuous chart parser, while covering 98

Differently, it is possible to rely on the idea that discontinuities are inherently related to the location of the token in the sentence. In this sense, it is possible to reorder the tokens while still obtaining a grammatical sentence that could be parsed by a continuous algorithm. This is usually achieved with transition-based parsing algorithms and the swap transition which switches the topmost elements in the stack. For instance, uses this transition to adapt an easy-first strategy for dependency parsing to discontinuous constituent parsing. In a similar vein, builds on top of a fast continuous shift-reduce constituent parser, and incorporates both standard and bundled swap transitions in order to analyze discontinuous constituents. system produces derivations of up to a length of n2 n + 1 given a sentence of length n. More efficiently, presents a transition system which replaces swap with a gap transition. The intuition is that a reduction does not need to be always applied locally to the two topmost elements in the stack, and that those two items can be connected, despite the existence of a gap between them, using non-local reductions. Their algorithm ensures an upper-bound of n(n1)2 transitions. With a different optimization goal, removed the traditional reliance of discontinuous parsers on averaged perceptrons and hand-crafted features for a recursive neural network approach that guides a swap-based system, with the capacity to generate contextualized representations. replace the stack used in transition-based systems with a memory set containing the created constituents. This model allows interactions between elements that are not adjacent, without the swap transition, to create a new (discontinuous) constituent. Trained on a 2 stacked BiLSTM transducer, the model is guaranteed to build a tree with in 4n-2 transitions, given a sentence of length n.

A middle ground between explicit constituent parsing algorithms and this paper is the work based on transformations. For instance, convert constituent trees into a nonlinguistic dependency representation that is learned by a transition-based dependency parser, to then map its output back to a constituent tree. A similar approach is taken by, but they proposed a more compact representation that leads to a much reduced set of output labels. Other authors such as propose a two-step approach that approximates discontinuous structure trees by parsing context-free grammars with generative probabilistic models and transforming them to discontinuous ones. cast discontinuous phrase-structure parsing into a framework that jointly performs supertagging and non-projective dependency parsing by a reduction to the Generalized Maximum Spanning Arborescence problem. The recent work by can be also framed within this paradigm. They essentially adapt the work by and replace the averaged perceptron classifier with pointer networks, adressing

In this context, the closest work to ours is the reduction proposed by, who cast continuous constituent parsing as sequence labeling. In the next sections we build on top of their work and: (i) analyze why their approach cannot handle discontinuous phrases, (ii) extend it to handle such phenomena, and (iii) train functional sequence labeling discontinuous parsers.

## 3 Preliminaries

Let w = [w0, w1, ..., w|w|1] be an input sequence of tokens, and T|w| the set of (continuous) constituent trees for sequences of length |w|; define an encoding function : T|w| $\rightarrow$ L|w| to map continuous constituent trees into a sequence of labels of the same length as the input. Each label, li L, is composed of three components li = (ni, xi, ui):

• ni encodes the number of levels in the tree in common between a word wi and wi+1. To obtain a manageable output vocabulary space, ni is actually encoded as the difference ni ni1, with n1 = 0. We denote by abs(ni) the absolute number of levels represented by ni. i.e. the total levels in common shared between a word and its next one.

• xi represents the lowest non-terminal symbol shared between wi and wi+1 at level abs(ni).

• ui encodes a leaf unary chain, i.e. nonterminals that belong only to the path from the terminal wi to the root. Note that cannot encode this information in (ni, xi), as these components always represent common information between wi and wi+1.

Incompleteness for discontinuous phrase structures proved that  is complete and injective for continuous trees. However, it is easy to prove that its validity does not extend to discontinuous trees, by using a counterexample. Figure 3 shows a minimal discontinuous tree that cannot be correctly decoded.

The inability to encode discontinuities lies on the assumption that wi+1 will always be attached to a node belonging to the path from the root to wi (ni is then used to specify the location of that node in the path). This is always true in continuous trees, but not in discontinuous trees, as can be seen in Figure 3 where c is the child of a constituent that does not lie in the path from S to b.

## 4 Encoding nearly ordered permutations

Next, we fill this gap to address discontinuous parsing as sequence labeling. We will extend the encoding  to the set of discontinuous constituent trees, which we will call T|w|. The key to do this relies on a well-known property: a discontinuous tree t  T|w| can be represented as a continuous one using an in-order traversal that keeps track of the original indexes (e.g. the trees at the left and the right in Figure 4). We will call this tree the (canonical) continuous arrangement of t, (t)  T|w|.

Thus, if given an input sentence we can generate the position of every word as a terminal in (t), the existing encodings to predict continuous trees as sequence labeling could be applied on (t). In essence, this is learning to predict a permutation of w. As introduced in §2, the concept of location of a token is not a stranger in transition-based discontinuous parsing, where actions such as swap switch the position of two elements in order to create a discontinuous phrase. We instead propose to explore how to handle this problem in end-to-end sequence labeling fashion, without relying on any parsing structure nor a set of transitions.

Todo so, first we denote by $\tau : \{0, \ldots, |w| - 1\} \rightarrow \{0, \ldots, |w| - 1\}$ the permutation that maps the position $i$ of a given $w_i$ in $w$ into its position as a terminal node in $\omega(t)$. From this, one can derive $\tau^{-1}$, a function that encodes a permutation of $w$ in such a way that its phrase structure does not have crossing branches. For continuous trees, $\tau$ and $\tau^{-1}$ are identity permutations. Then, we extend the tree encoding function $\Phi$ to $T_{|w|} \rightarrow L'^{}_{|w|}$ where $l \in L'$ is enriched with a fourth component $p_i$ such that $l = (n_i, x_i, u_i, p_i)$, where $p_i$ is a discrete symbol such that the sequence of $p_i$'s encodes the permutation $\tau$ (typically, each $p_i$ will be an encoding of $\tau(i)$, i.e., the position of $w_i$ in the continuous arrangement, although this need not be true in all encodings, as will be seen below).

The crux of defining a viable encoding for discontinuous parsing is then in how we encode tau as a sequence of values pi, for i = 0 . . . |w|  1. While the naive approach would be the identity encoding (pi = tau(i)), we ideally want an encoding that balances minimizing sparsity (by minimizing infrequently-used values) and maximizing learnability (by being predictable). To do so, we will look for encodings that take advantage of the fact that discontinuities in attested syntactic structures are mild , i.e., in most cases, tau (i + 1) = tau (i) + 1. In other words, permutations tau corresponding to real syntactic trees tend to be nearly ordered permutations. Based on these principles, we propose below a set of concrete encodings, which are also depicted on an example in Figure 4. All of them handle multiple gaps (a discontinuity inside a discontinuity) and cover 100

**Absolute-position:** For every token $w_i$, $p_i = \tau(i)$ only if $w_i \neq \tau(i)$. Otherwise, we use a special label `INV`, which represents that the word is a fixed point in the permutation, i.e., it occupies the same place in the sentence and in the continuous arrangement.

Relative-position If i != tau(i), then pi = i  tau(i). otherwise, we again use the INV label.

Lehmer code In combinatorics, let n = [0, ..., n  1] be a sorted sequence of objects, a Lehmer code is a sequence sigma = [sigma0, ...sigman1] that encodes one of the n! permutations of n, namely . The idea is intuitive: let ni+1 be the subsequence of objects from n that remain available after we have permuted the first i objects to achieve the permutation , then sigmai+1 equals the (zero-based) position in ni+1 of the next object to be selected. For instance, given n = [0, 1, 2, 3, 4] and a valid permutation  = [0, 1, 3, 4, 2], then sigma = [0, 0, 1, 1, 0]. Note that the identity permutation would be encoded as a sequence of zeros.

In the context of discontinuous parsing and encoding pi, n can be seen as the input sentence w where pi(w) is encoded by sigma. The Lehmer code is particularly suitable for this task in terms of compression, as in most of the cases we expect (nearly) ordered permutations, which translates into the majority of elements of sigma being zero. However, this encoding poses some potential

| Label Component | TIGER | Labels NEGRA | DPTB |
|---|---|---|---|
| ni | 22 | 19 | 34 |
| ti | 93 | 56 | 137 |
| ui | 15 | 4 | 56 |
| pi as absolute-position | 129 | 110 | 98 |
| pi as relative-position | 105 | 90 | 87 |
| pi as Lehmer | 39 | 34 | 27 |
| pi as inverse Lehmer | 68 | 57 | 61 |
| pi as pointer-based | 122 | 99* | 110* |
| pi as pointer-based simplified | 81 | 65 | 83* |

Table 1: Number of values per label component, merging the training and dev sets (gold setup). *are codes that generate one extra label with predicted PoS tags (this variability depends on the used PoS-tagger).

| Hyperparameter | Value |
|---|---|
| BiLSTM size | 800 |
| # BiLSTM layers | 2 |
| optimizer | SGD |
| loss | cat. cross-entropy |
| learning rate | 0.2 |
| decay (linear) | 0.05 |
| momentum | 0.9 |
| dropout | 0.5 |
| word embs | Ling et al. (2015) |
| PoS tags emb size | 20 |
| character emb size | 30 |
| batch size training | 8 |
| training epochs | 100 |
| batch size test | 128 |

Table 2: Main hyper-parameters for the training of the BiLSTMs, both for the gold and predicted setups

learnability problems. The root of the problem is that sigmai does not necessarily encode tau(i), but tau(j) where j is the index of the word that occupies the ith position in the continuous arrangement (i.e., j = tau 1(i)). In other words, this encoding is expressed following the order of words in the continuous arrangement rather than the input order, causing a non-straightforward mapping between input words and labels. For instance, in the previous example, sigma2 does not encode the location of the object n2 = 2 but that of n3 = 3.

Lehmer code of the inverse permutation To ensure that each pi encodes tau(i), we instead interpret pi as meaning that should fill the (pi + 1)th currently remaining blank in a sequence sigma that is initialized as a sequence of blanks, i.e. sigma = [,,...,] $.For instance, letn = [0, 1, 2, 3, 4] be$

Pointer-based encoding When encoding tau(i), the previous encodings generate the position for the target word, but they do not really take into account the left-to-right order in which sentences are naturally read, nor they are linguistically inspired. In particular, informally speaking, in human lin-

Finally, in Table 11 we list the number of parameters for each of the transducers trained on the pointer-based encoding. For the rest of the encodings, the models have a similar number of parameters, as the only change in the architecture is the small part involving the feed-forward output layer that predicts the label component pi.

More in detail, for BiLSTMs and vanilla Trans-

formers, the word embeddings are pre-trained FastText embeddings with 100 dimensions for English and 60 for German, and the PoS tags are represented by an embedding layer of 20 dimensions.

| Hyperparameter | Value (gold setup) | Value (pred setup) |
|---|---|---|
| Att. heads | 8 | 8 |
| Att. layers | 6 | 6 |
| Hidden size | 800 | 800 |
| Hidden dropout | 0.4 | 0.4 |
| optimizer | SGD | SGD |
| loss | Cross-entropy | Cross-entropy |
| learning rate | 0.004* | 0.003 |
| decay (linear) | 0.0 | 0.0 |
| momentum | 0.0 | 0.0 |
| word embs | Previous Works | |
| PoS tags emb size | 20 | 20 |
| character emb size | 136/132batch size training | 8 |
| 8 | | |
| training epochs | 400 | 400 |
| batch size test | 128 | 128 |

Table 3: Main hyper-parameters for training the Transformer encoders

| Model | Parameters |
|---|---|
| Pointer-based BiLSTM | 13.9 M |
| Pointer-based Transformer | 23.4 M |
| Pointer-based DistilBERT | 73 M |
| Pointer-based BERT base | 108 M |
| Pointer-based BERT large | 330 M |

Table 4: Number of parameters per model.

Additionally we use a char-based LSTM with a hidden layer of 100/132 dimensions (English/German). For both approaches, a linear layer followed by a softmax is used to predict every label component.

For BERT and DistilBERT we use the default fine-tuning parameters. We use Adam as optimizer and cross entropy as the loss function. The learning rate and other hyper-parameters are left as default in the transformers library, except for the number of training epochs (we train them for at most 30 epochs), and the batch size, which is adjusted depending on the memory required by the model (e.g. 8 for BERT and 32 for DistilBERT). For the BERT-large model, due to the limitations in GPU memory, we have to reduce the training batch size to 1, and use a smaller learning rate of 1e-5.

## 5 Experiments

Setup For English, we use the discontinuous Penn Treebank (DPTB) by. For German, we use TIGER and NEGRA. We use the splits by which in turn follow the splits for the NEGRA treebank, the splits for TIGER, and the standard splits for (D)PTB (Sections 2 to 21 for training, 22 for development and 23 for testing). See also Appendix A.5 for more detailed statistics. We consider gold and predicted PoS tags. For the latter, the parsers are trained on predicted PoS tags, which are generated by a 2stacked BiLSTM, with the hyper-parameters used to train the parsers. The PoS tagging accuracy (

Metrics We report the F-1 labeled bracketing score for all and discontinuous constituents, using discodop and the proper.prm parameter file. Model selection is based on overall bracketing F1score.

### 5.1 Results

Table 2 shows the results on the dev sets for all encodings and transducers. The tendency is clear showing that the pointer-based encodings obtain the best results. The pointer-based encoding with simplified PoS tags does not lead however to clear improvements, suggesting that the models can learn the sparser original PoS tags set. For the rest of encodings we also observe interesting tendencies. For instance, when running experiments using stacked BiLSTMs, the relative encoding performs better

than the absolute one, which was somehow expected as the encoding is less sparse. However, the tendency is the opposite for the Transformer encoders (including BERT and DistilBERT), especially for the case of discontinuous constituents. We hypothesize this is due to the capacity of Transformers to attend to every other word through multihead attention, which might give an advantage to encode absolute positions over BiLSTMs, where the whole left and right context is represented by a single vector. With respect to the Lehmer and Lehmer of the inverse permutation encodings, the latter performs better overall, confirming the bigger difficulties for the tested sequence labelers to learn Lehmer, which in some cases has a performance even close to the naive absolute-positional encoding (e.g. for TIGER using the vanilla Transformer encoder and BERT). As introduced in §4, we hypothesize this is caused by the non-straightforward mapping between words and labels (in the Lehmer code the label generated for a word does not necessarily contain information about the position of such word in the continuous arrangement).

In Table 3 we compare a selection of our models against previous work using both gold and predicted PoS tags. In particular, we include: (i) models using the pointer-based encoding, since they obtained the overall best performance on the dev sets, and (ii) a representative subset of encodings (the absolute positional one and the Lehmer code of the inverse permutation) trained with the best performing transducer. Additionally, for the case of the (English) DPTB, we also include experiments using a bert-large model, to shed more light on whether the size of the networks is playing a role when it comes to detect discontinuities. Additionally, we report speeds on CPU and GPU. The experiments show that the encodings are learnable, but that the model's power makes a difference. For instance, in the predicted setup BILSTMs and vanilla Transformers perform in line with predeep learning models , DistilBERT already achieves a robust performance, close to models such as and BERT transducers suffice to achieve results close to some of the strongest approaches, e.g.. Yet, the results lag behind the state of the art. With respect to the architectures that performed the best the main issue is that they are the bottleneck of the pipeline. Thus, the computation of the contextualized word vectors under current approaches greatly decreases the importance, when it comes to speed, of the chosen parsing paradigm used to generate the output trees (e.g. chart-based versus sequence labeling).

Finally, Table 4 details the discontinuous performance of our best performing models.

Discussion on other applications It is worth noting that while we focused on parsing as sequence labeling, encoding syntactic trees as labels is useful to straightforwardly feed syntactic information to downstream models, even if the trees themselves come from a non-sequence-labeling parser. For example, use the sequence labeling encoding of to provide syntactic information to a semantic role labeling model. Apart from providing fast and accurate parsers, our encodings can be used to do the same with discontinuous syntax.

## 6 Conclusion

We reduced discontinuous parsing to sequence labeling. The key contribution consisted in predicting a continuous tree with a rearrangement of the leaf nodes to shape discontinuities, and defining various ways to encode such a rearrangement as a sequence of labels associated to each word, taking advantage of the fact that in practice they are nearly ordered permutations. We tested whether those encodings are learnable by neural models and saw that the choice of permutation encoding is not trivial, and there are interactions between encodings