Hierarchical Bracketing Encodings for Dependency Parsing as Tagging

Ana Ezquerro[1] ana.ezquerro
David Vilares[1] david.vilares
Anssi Yli-Jyrä[2,3] anssi.yli-jyra
Carlos Gómez-Rodríguez[1] carlos.gomez

[1]Universidade da Coruña, CITIC (@udc.es)
[2]Tampere University, Math. Res. Centre, Computing Sciences (@tuni.fi)
[3]University of Helsinki, Faculty of Arts, Digital Humanities (@helsinki.fi)

Abstract

We present a family of encodings for sequence labeling dependency parsing, based on the concept of hierarchical bracketing. We prove that the existing 4-bit projective encoding belongs to this family, but it is suboptimal in the number of labels used to encode a tree. We derive an optimal hierarchical bracketing, which minimizes the number of symbols used and encodes projective trees using only 12 distinct labels (vs. 16 for the 4-bit encoding). We also extend optimal hierarchical bracketing to support arbitrary non-projectivity in a more compact way than previous encodings. Our new encodings yield competitive accuracy on a diverse set of treebanks.

1 Introduction

Sequence labeling with contextualized representations enables efficient dependency parsers that function as taggers while achieving high or even state-of-the-art performance (Amini et al., 2023; Gómez-Rodríguez et al., 2023). This paradigm has two key aspects: a powerful sequence encoder—often a pre-trained model—and a linearization algorithm that transforms dependency trees into label sequences with a strict one-to-one mapping. Regarding linearizations, various strategies have been proposed. Early approaches represented a word's head as an offset (Strzyz et al., 2019; Lacroix, 2019), which could be absolute, relative to the dependent, or based on word properties. Another strategy is bracketing encodings (Yli-Jyrä and Gómez-Rodríguez, 2017; Strzyz et al., 2019), where each word label encodes a set of incoming or outgoing arcs, unlike positional encodings that explicitly encode the head. A third approach, transition-based encodings (Gómez-Rodríguez et al., 2020), assigns labels to transition subsequences split by read (shift-like) transitions.

These strategies suffer from an unbounded label space. Instead, recent work proposes fixed-space solutions. Amini et al. (2023) proposed hexatagging, a projective tree linearization with eight labels but requiring an intermediate representation. Gómez-Rodríguez et al. (2023) introduced 4- and 7-bit labels for projective and 2-planar trees, with bounded label spaces of 16 and 128 labels.

This work introduces a new family of hierarchical bracketing encodings, based on rope cover (Yli-Jyrä, 2019b). We show that the existing 4-bit projective encoding is part of this family, but suboptimal in terms of compactness. We introduce and test its optimal counterpart, and show that both bracketings improve accuracy for non-projective trees when the pseudo-projective transformation is applied. We also generalize the optimal bracketing to directly encode arbitrary non-projective trees.

2 Preliminaries

Let $w_1 \dots w_n$ be an input string. A dependency graph is a directed graph $G = (V, E)$ where $V = \{0, \dots, n\}$. An edge (i, j) in E is called a dependency from word w_i to w_j. w_i is the head or parent, w_j is the dependent. Index 0 is used for a dummy node used when we deal with trees. We use the notation $i \rightarrow j$ to specify that such a dependency is a rightward arc (i.e., when $i < j$) and $i \leftarrow j$ when it is a leftward arc ($i > j$). When direction is not specified, we use (i, j). Node k is a descendant of node i if there is a path from i to k.

An arc (i, j) is said to *cross* the arc (k, l) if $\min(i, j) < \min(k, l) < \max(i, j) < \max(k, l)$ or $\min(k, l) < \min(i, j) < \max(k, l) < \max(i, j)$. An arc (i, j) is said to *cover* arc (k, l) if $\min(i, j) \leq \min(k, l) < \max(k, l) \leq \max(i, j)$.

A dependency graph is a tree if it has no cycles and every node has exactly one parent except for the dummy node 0, which has no parent. A tree is projective if it has no crossing arcs. An equivalent definition (see Nivre (2006)) will be more directly useful in some proofs: a dependency tree is projective if, for each arc (i, j), all nodes located between i and j are descendants of either i or j.

3 Brackets and superbrackets

3.1 Bracketing encodings

Bracketing encodings for dependency parsing are based on the idea of representing dependencies using symbols that behave as balanced brackets. In the earliest and most basic bracketing encoding (Strzyz et al., 2019, 2020), a right arc from a word w_i to w_j is encoded by including an opening bracket / in the label of w_i, which matches the corresponding closing bracket > in the label of w_j. Similarly, a left arc from w_j to w_i is represented by an opening bracket < in the label of w_i that matches a closing bracket \ in the label of w_j. The result is shown in Figure 1a, where the label associated with each given word contains one < (resp. >) symbol per incoming arc from the right (left) and one / (resp. \) symbol per outgoing arc towards the right (left). To decode the brackets back into a tree, we read them from left to right, using a stack. When reading an opening bracket (/ or <) in the label of a word w_i, we push it to the stack, associating it with the index i. When reading a closing bracket > in the label of w_j, we pop the matching bracket / from the top of the stack, check its associated index i, and create the right arc from w_i to w_j. Finally, when reading a closing bracket \, we proceed analogously, popping a < from the top of the stack and creating a left arc. This decoding method will recover the original tree if it is *projective* (i.e., without crossing arcs), as the nested matching of brackets prevents them from being paired in any way that would create crossing arcs.[1] Extensions to support crossing arcs exist (Strzyz et al., 2020), but we will stick to projective trees in this section.

This encoding obtains good practical results, especially in low-resource setups (Muñoz-Ortiz et al., 2021). Yet, it has the disadvantage that it is *unbounded* in terms of label set size, as it scales with sentence length. In particular, the label set size for projective trees is $\Theta(n^2)$, as a label can have any combination of $\Theta(n)$ \ symbols and $\Theta(n)$ / symbols adding up to at most $n - 1$ total such symbols, denoting outgoing left and right arcs. In contrast, newer bounded encodings like hexatagging (Amini et al., 2023) use a constant number of labels. To address this, Gómez-Rodríguez et al. (2023) proposed the 4-bit encoding, a compact bracketing scheme for projective trees with 16 fixed labels.

We next define a framework to compress bracketing encodings, based on Yli-Jyrä (2019b)'s notion of *rope cover*, of which we will show that the 4-bit encoding is a particular case. However, the 4-bit encoding is not optimal in bracket usage, so we propose an encoding that represents any projective tree with 12 labels instead of 16.

3.2 Hierarchical bracketing framework

To avoid unbounded strings of brackets in dependency labels, we establish a hierarchy of two levels of dependency arcs, yielding a hierarchical bracketing scheme with two levels of brackets: superbrackets and semibrackets (Yli-Jyrä, 2019b,a).

We first illustrate this by example. Consider the dependency tree in Figure 1a. We observe that the leftmost part (nodes from 0 to 4) is composed of an arc $0 \to 4$ and a number of arcs that go inwards from its two endpoints. Taking advantage of this, we declare $0 \to 4$ to be a *structural arc*, and we encode it with two *superbrackets* **/** and **>** (with the complete set of superbrackets being $\{$**<**, **/**, \, **>**$\}$).

All the other arcs between nodes 0 and 4 *lean on* the structural arc $0 \to 4$, in the sense that they are covered by it and they share one of its endpoints. We call them *auxiliary arcs* associated with that structural arc. Exploiting this, we encode each of these arcs with a single symbol, called a *semibracket*, which we place in the label of the endpoint that is not shared with the structural arc. For example, to encode the left arc $3 \leftarrow 4$, we just place a left semibracket < in the label of node 3, indicating that it has an incoming auxiliary arc from the right. A bracket at the other endpoint (4) is not needed, because the arc, being auxiliary, is supposed to lean on a structural arc. Thus, the right endpoint is understood to be at the nearest closing superbracket to the right (note that it does not matter whether this superbracket is of type \ or **>**). Analogously, the arc $0 \to 1$ only needs a semibracket > with the left endpoint being the closest opening superbracket to the left, in this case **/**.

This leads to the encoding in Figure 1b. The nodes of the second part of the tree (4 to 7) are assigned with super and semibrackets following a similar procedure. The auxiliary arc $6 \leftarrow 7$ leans

[1]The original implementation in (Strzyz et al., 2019, 2020) does not decode like this. It decodes using two separate stacks for left and right arcs, and then the coverage is extended to support crossing arcs in opposite directions. Here we assume projective decoding with a single stack for didactic reasons, as it helps us establish the relation to our novel encodings. Extensions for non-projectivity will be presented later.

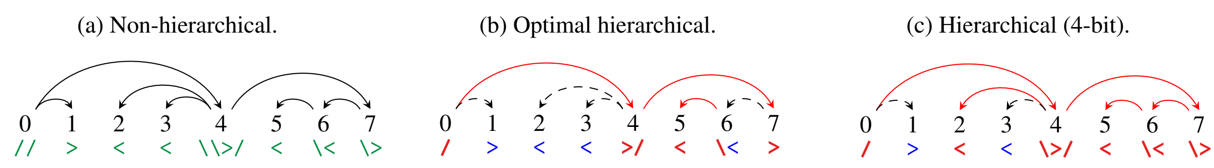

Figure 1: Example showing the same tree encoded with the standard (non-hierarchical) bracketing encoding (Figure 1a), and hierarchical bracketing encodings (Figures 1b and 1c). In the latter, structural arcs are shown in red and solid, whereas auxiliary arcs are black and dashed. Figure 1b corresponds to the optimal hierarchical bracketing where $0 \rightarrow 4$, $4 \rightarrow 7$ and $6 \rightarrow 5$ are the structural arcs. Figure 1c corresponds to the 4-bit encoding.

on the structural arc $4 \rightarrow 7$, just like $3 \leftarrow 4$ did on $0 \rightarrow 4$, so $4 \rightarrow 7$ is also encoded with / and > while $6 \leftarrow 7$ only needs < in node 6, with the right endpoint being given by > in node 7. The arc $5 \leftarrow 6$ does not lean on any other arc of the tree, so we mark it as another structural arc.

As observed, this means that we can encode the tree with fewer brackets than the original bracketing encoding (10 vs. 14, cf. Figures 1a and 1b counting the number of individual bracket symbols). The amount of brackets saved depends on which arcs we mark as structural, as each structural arc needs two brackets while auxiliary arcs are represented with just one. For example, Figure 1c shows an alternative where two more arcs are marked as structural, leading to using two more brackets. Thus, minimizing the number of brackets needed requires finding the minimum possible set of structural arcs such that all other arcs lean on them. This yields the *optimal hierarchical bracketing*, which in the example is the one in the center.

Formalization We now define the framework, following the definitions of Yli-Jyrä (2019b).[2] We say that an arc (i, j) *supports* (k, l) (or equivalently, (k, l) *leans* on (i, j)) if (i, j) covers (k, l) and either $\min(i, j) = \min(k, l)$ or $\max(i, j) = \max(k, l)$. For example, in the tree in Figure 1, $0 \rightarrow 4$ supports $0 \rightarrow 1$, $2 \leftarrow 4$ and $3 \leftarrow 4$.

A subset $R \subseteq E$ is a *rope cover* of a dependency graph $G = (V, E)$ if every arc in $E \setminus R$ leans on at least one arc in R. Given such a rope cover, we call the arcs in R *structural arcs*, and the arcs in $E \setminus R$ *auxiliary arcs*. R is said to be a *proper rope cover* if no arc in R leans on another arc of R. Yli-Jyrä (2019b) proves that the proper rope cover of a graph is unique.[3] In Figures 1b and 1c, the subsets of red, solid arcs are rope covers of the example tree, with the one in 1b being the proper rope cover.

Encoding Given a dependency graph $G = (V, E)$ where no arcs cross, and a rope cover $R \subseteq E$, we can encode the graph as follows (and we call this encoding the *hierarchical bracketing encoding of G induced by R*): for each rightward arc $w_i \rightarrow w_j \in R$, we add an opening superbracket / to the label of w_i and a closing superbracket > to the label of w_j. For each leftward arc $w_i \leftarrow w_j \in R$, we add an opening superbracket < to the label of w_i and a closing superbracket \ to the label of w_j. For an arc $(w_i, w_j) \notin R$, if it leans on a structural arc with left (right) endpoint w_i, we add to the label of w_j the semibracket > (<). Otherwise, it will lean on a structural arc with left (right) endpoint w_j, and we add to the label of w_i the semibracket \ (/). Regarding the ordering of brackets within the same label, we always arrange them in such a way that closing brackets appear first (in increasing order of arc length) followed by opening brackets (in decreasing order of arc length), to ensure correct nesting.

Decoding To decode, we use Algorithm 1. The algorithm uses a stack S to process brackets and put the resulting arcs into the arc set A. Labels and symbols therein are read from left to right. When an opening bracket (of any kind) is found, it is pushed to the stack, keeping track of the index of the label where it appeared. When a closing semibracket is read, it is matched to the top node on

[2]Due to differences in scope, we do not literally follow the terminology of Yli-Jyrä (2019b), but make a substantive adaptation for our purposes. We skip definitions that we do not need, add some extra terms (most importantly structural and auxiliary arcs, which are present in (Yli-Jyrä, 2019b) but were not given an explicit name), change the perspective on others, simplify the bracketing notation, and consider non-optimal bracketings which were not contemplated by Yli-Jyrä.

[3]There is a specific case where it is not unique: graphs with a cycle of length 2, i.e. arcs $a \rightarrow b$ and $a \leftarrow b$, where either could be structural yielding two proper rope covers. This case is overlooked in (Yli-Jyrä, 2019b) as the proof is based on the underlying undirected graph of the dependency graph. However, this cannot happen in trees, and even in graphs, it can easily be solved by setting an arbitrary criterion in the definition (i.e., to force the right arc to be the structural one in this situation). Thus, we ignore this exception from now on.

the stack (which can be any opening superbracket), creating the corresponding arc, but without removing said superbracket from the stack. Finally, when a closing superbracket is found, it is matched to any semibrackets on top of the stack until finding the first superbracket, which is also matched. All these matched nodes are removed from the stack, and the corresponding arcs created.

Algorithm 1 Decoding for Noncrossing Graphs

```
 1: procedure DECODE($l_1, \dots, l_n$)
 2:     $S \leftarrow$ empty stack ; $A \leftarrow$ empty set
 3:     for $i = 1 \rightarrow n$ do
 4:         for symbol in $l_i$ do
 5:             if symbol $\in \{<, /, <, /\}$ then
 6:                 PUSH($S$, (symbol, $i$))
 7:             else if symbol $=$ > then
 8:                 $(s, j) \leftarrow$ PEEK($S$)          ▷ $s \in \{/, <\}$
 9:                 $A \leftarrow A \cup \{j \rightarrow i\}$
10:             else if symbol $=$ \ then
11:                 $(s, j) \leftarrow$ PEEK($S$)          ▷ $s \in \{/, <\}$
12:                 $A \leftarrow A \cup \{j \leftarrow i\}$
13:             else if symbol $=$ > or symbol $=$ \ then
14:                 CLOSESUPERBRACKET(symbol, i, S, A)

15: procedure CLOSESUPERBRACKET(rbracket, i, S, A)
16:     (lbracket, $j$) $\leftarrow$ POP($S$)
17:     while lbracket $=$ < or lbracket $=$ / do
18:         if lbracket $=$ < then $A \leftarrow A \cup \{j \leftarrow i\}$
19:         else $A \leftarrow A \cup \{j \rightarrow i\}$          ▷ lbracket $=$ /
20:         (lbracket, $j$) $\leftarrow$ POP($S$)
21:     if lbracket $=$ < and rbracket $=$ \ then
22:         $A \leftarrow A \cup \{j \leftarrow i\}$
23:     else if lbracket $=$ / and rbracket $=$ > then
24:         $A \leftarrow A \cup \{j \rightarrow i\}$
```

The complexity of Algorithm 1 is linear with respect to the number of arcs in the graph. This can be shown by observing that, for each arc, we perform exactly one push to the stack (when the left bracket is read); as well as exactly one pop or one peek, depending on whether the right bracket is a superbracket or a semibracket. Thus, when operating on dependency trees, complexity with respect to sentence length is $O(n)$, as there is one arc per word; whereas for dense graphs (not often found in NLP) complexity can increase to $O(n^2)$.

Ensuring well-formedness Like all parsing-as-sequence-labeling approaches, hierarchical bracketing encodings are not surjective, so practical decoding implementations need to be able to deal with ill-formed label sequences (for example, those with mismatched brackets). In our implementation of Algorithm 1, we follow common practice in sequence labeling parsing, by which illegal label sequences are handled by very simple heuristics. In particular, if brackets fail to match, we just match matching brackets and ignore any extra brackets – unclosed opening brackets remain in the stack after Algorithm 1 and not create any arcs, and unmatched closing brackets (e.g. when we try to pop the stack in line 16 or 20, but it's empty) are discarded. If the desired output is a tree, cycles and reentrancies are avoided by checking for them whenever the algorithm adds a new arc to A (which can be done in inverse Ackermann time using path compression and union by rank, i.e., it does not affect the computational complexity of the decoding algorithm) and nodes without a parent are attached to the syntactic root in post-processing.

Compact hierarchical bracketing The hierarchical bracketing framework we have just introduced can be used to define different encodings, depending on which criteria we use to set the structural arcs (i.e., to obtain a rope cover). In the worst case, the resulting encodings can still be unbounded: for example, if we consider the degenerate case of a rope cover where $R = E$ (i.e., all arcs are structural), the resulting encoding is equivalent to the naive bracketing of (Strzyz et al., 2019, 2020).

We now define a sufficient condition to obtain bounded encodings. Given a dependency graph $G = (V, E)$, and a rope cover $R \subseteq E$, we say that R is a *compact rope cover* if no node in V is a head of more than one structural arc going to the same direction, and no node is a dependent of more than one structural arc coming from the same direction. This condition limits the amount of superbrackets of each kind that can appear in the same label to 1 (e.g., we cannot have two / on a label because we cannot have more than one outgoing structural arc to the right). Since the amount of semibrackets of each kind that can appear in the same label is always at most 1 regardless of the rope cover used (semibrackets always match the closest superbracket and do not modify the stack, so a group of equal semibrackets would all create the same arc); a compact rope cover will induce an encoding where the length of each label is bounded by a constant, which is thus bounded. We call the encoding induced by a compact rope cover a *compact hierarchical bracketing encoding*.

3.3 Properties of hierarchical brackets for projective trees

The framework described in Section 3.2 works for any dependency graph without crossing arcs. However, it is especially interesting to focus on projective dependency trees, as their properties guarantee a particularly compact label set. In particular, we can show the following result:

Theorem 1 *Any compact hierarchical bracketing encoding for projective dependency trees uses at most 16 labels.*

To prove it, we consider the following:

- Since each node in a dependency tree has exactly one parent, each label for a dependency tree (projective or not) must have exactly one bracket from the set {<, <, >, >}.
- The encoding for a projective dependency tree cannot have any brackets of the form \ or /. \ can only appear if a leftward arc leans on an arc that covers it. In a tree, the covering arc must necessarily be a rightward arc (so the single-head constraint is not violated) and this means that the node with the bracket \ dominates over the covering arc, which cannot happen in a projective tree. The same reasoning (but mirrored) applies to /.
- We already established in the general framework that closing brackets appear before opening brackets in labels. In projective trees, we can further observe that leftward closing brackets (\) must appear before rightward ones (> or >), and leftward opening brackets (< or <) before rightward ones (/). This is for similar reasons as the previous point: for example, a label with > or > before \ would mean that the corresponding node has an incoming arc from the left and a *longer* outgoing arc to the right. In the context of a tree, this means that the node dominates an arc that covers it, which is forbidden in projective trees. Analogous reasoning can be done for the other forbidden combinations.

Taking all these points into account, we conclude that the labels of a compact hierarchical bracketing encoding for projective dependency trees must necessarily be of the form (\)? (> | > | < | <) (/)?

Counting the labels allowed by this expression, we directly derive Theorem 1.

3.4 The 4-bit encoding as hierarchical bracketing

Gómez-Rodríguez et al. (2023) present the *4-bit encoding*: a 16-label encoding for a superset of projective dependency trees, where each label is composed of 4 bits ($b_0b_1b_2b_3$): b_0 indicates whether the corresponding node is a left or right dependent, b_1 whether it is the farthest left or right dependent, and b_2 (b_3) whether it has any left (right) dependents.

b_0b_1 \ b_2b_3	00	01	10	11
00	<	< /	\ <	\ < /
01	<	< /	\ <	\ < /
10	>	> /	\ >	\ > /
11	>	> /	\ >	\ > /

Table 1: Conversion of the 4-bit labels (Gómez-Rodríguez et al., 2023) to hierarchical bracketing labels.

This encoding can be seen as a compact hierarchical bracketing for projective trees. To see it, for a given projective tree $T = (V, E)$, consider the rope cover R_{4b} built by taking the longest rightward arc and longest leftward arc going out of each node. Under this rope cover, the bracket from the set {<, <, >, >} associated to a given word will be a superbracket if, and only if, that word is the farthest left or right dependent of its head (i.e., if $b_1 = 1$). Since b_0 determines direction, b_0 and b_1 together select exactly one bracket from that set. In turn, presence of \ in a label corresponds to $b_2 = 1$ (indicating having left dependents, which means we must use \ for the origin of the corresponding left arc(s) since \ cannot appear in projective trees, as argued earlier) and the same applies to / and $b_3 = 1$. Thus, 4-bit labels are isomorphic to the labels of the compact hierarchical bracketing encoding induced by the rope cover R_{4b}, with the correspondence shown in Table 1. A simple example of a tree with its rope cover R_{4b} and the induced labeling, corresponding to the 4-bit encoding, is shown in Figure 1c.

3.5 Optimal hierarchical bracketing

By viewing the 4-bit encoding through the lens of the hierarchical bracketing framework, we can notice that it is suboptimal in terms of number of brackets needed to encode a tree. An example can be seen in Figure 1, where the 4-bit encoding yields the rope cover and labels shown on the right; but the alternative in the center has a rope cover with two fewer structural arcs (and thus, uses two fewer brackets) to encode the same tree. This begs the question of whether it is possible to define an *optimal hierarchical bracketing* for projective trees, i.e. one that guarantees encoding each tree with the minimum possible amount of brackets within hierarchical bracketing encodings.

This can be done by computing the proper rope cover. Yli-Jyrä (2019b) provides a simple algorithm for this purpose, which works for any dependency graph $G = (V, E)$:

1. All arcs in E start unmarked. Among the unmarked arcs with the leftmost left endpoint, take the longest one and mark it as structural arc, thus adding it to the proper rope cover.
2. Mark all the arcs that lean on it as auxiliary.
3. Repeat the process until every arc is marked.

We will now show the following previously unproven property of the proper rope cover:

Theorem 2 *The proper rope cover of a dependency graph $G = (V, E)$ has minimum cardinality among rope covers of G.*

To prove this, we can proceed as follows. Let R_{pr} be the unique proper rope cover of G. We claim that R_{pr} must be of minimum cardinality among all rope covers. Equivalently, we can show that *any* rope cover $R \subseteq E$ can be transformed into some rope cover R' (possibly itself) that is proper with $|R'| \leq |R|$. Since the proper rope cover is unique, it follows $R' = R_{pr}$, forcing $|R_{pr}| \leq |R|$. Hence $|R_{pr}|$ is minimal.

The transformation process is as follows. If R is initially proper, the transformation is done. If it is not proper, then by definition there exist arcs $r_1, r_2 \in R$ such that r_1 leans on r_2. Choose such a pair of arcs. By definition, r_1 and r_2 share an endpoint e, and r_2 covers r_1. Without loss of generality, suppose that e is the right endpoint of r_1 and r_2. We call e_l the left endpoint of r_1.

We will now remove r_1 from the rope cover, possibly replacing it with another arc. To do this, we need to take into account that all arcs in $E \setminus R$ that lean on r_1 need to still be supported by a structural arc in the new rope cover. Arcs can be in this situation either because their right endpoint is e, or their left endpoint is e_l. The former are supported by r_2, and will continue to be even if we remove r_1. So we only need to ensure that any arcs with left endpoint e_1 (and covered by r_1) have support. For this purpose, if there are arcs in this situation, we take the longest and add it to the rope cover to replace r_1 (if it was not already in the rope cover). This arc will cover the remaining arcs,

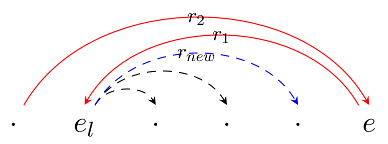

Figure 2: An iteration of the process to transform a rope cover to the proper rope cover. The arc r_1 in the rope cover would be replaced with r_{new}.

so the obtained arc set is a new rope cover. This process is illustrated graphically in Figure 2.

This process either preserves the size of the rope cover (if r_1 is replaced with another arc) or decreases it by 1. Also note that it strictly reduces the sum of edge lengths of the rope cover (even if no arc is removed, r_1 is replaced with a strictly shorter arc covered by it). Thus, we can iterate it while there is a pair of leaning arcs, with the assurance that it will finish, and the result will be the proper rope cover. This concludes the proof of Theorem 2.

From this, we can derive the following result:

Corollary 1 *The hierarchical bracketing induced by the proper rope cover is optimal.*

This is shown by observing that, in the hierarchical bracketing encoding induced by a rope cover, each structural arc generates two superbrackets, whereas each auxiliary arc generates one semibracket. By Theorem 2, the proper rope cover has the minimum possible amount of structural arcs, thus, it also minimizes the total number of brackets generated for a given graph.

Hence, with the above algorithm by Yli-Jyrä (2019b) to compute the proper rope cover, we get a hierarchical bracketing which is optimal in terms of number of brackets used to encode each given tree. We will now prove that, when applied to projective trees, it also reduces the number of labels with respect to alternatives like e.g. the 4-bit encoding:

Theorem 3 *When applied to projective trees, the optimal hierarchical bracketing encoding induced by the proper rope cover uses at most 12 labels.*

To prove this, consider the 16 possible labels of compact hierarchical bracketings (Theorem 1), which are also shown explicitly in Table 1 (since they are also the labels for the 4-bit encoding). Out of those 16 labels, there are four that cannot appear in the bracketing induced by the proper rope cover: `</`, `\>`, `\</` and `\>/`. This is because presence of a label including `</` indicates that its corresponding node has both an incoming structural arc from the

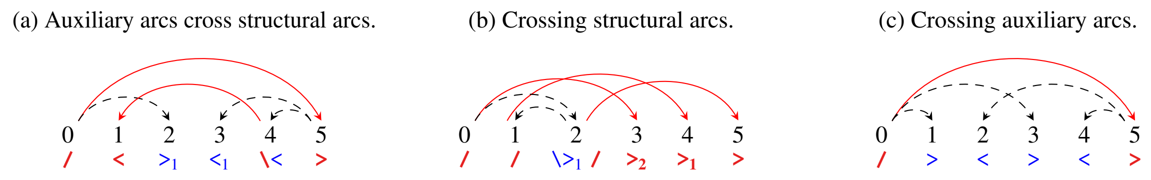
(a) Auxiliary arcs cross structural arcs.
0 1 2 3 4 5
/ < >1 <1 \< >
(b) Crossing structural arcs.
0 1 2 3 4 5
/ / \>1 / >2 >1 >
(c) Crossing auxiliary arcs.
0 1 2 3 4 5
/ > < > < >

Figure 3: Examples of the non-projective extension. In Figure 3a, we add indexes to semibrackets to skip the matching superbracket. In Figure 3b, indexes are also added to superbrackets to skip multiple opening superbrackets. In Figure 3c, no indexes are needed when auxiliary arcs cross other auxiliary arcs.

right and an outgoing structural arc to the right and, by construction, one would lean on the other, which is not allowed in the proper rope cover. The same reasoning can be made for the combination \>.

Thus, we have derived a hierarchical bracketing encoding for projective trees that is easily obtainable and reduces the label set from 16 to 12 when compared with the 4-bit encoding, while sharing common underlying principles.

4 Non-projectivity

Our above definitions of hierarchical bracketing are for dependency graphs without crossing arcs. We now add the possibility of crossing arcs (and thus support non-projective trees). One option to do so would be to use multiplanarity (dividing the graph into subsets without crossings) as in (Strzyz et al., 2020; Gómez-Rodríguez et al., 2023). We choose not to follow this path here as it is not a straightforward match with our single-stack decoding process, and instead, again inspired by Yli-Jyrä (2019b), we use brackets that can skip others when matching.

We make the next extensions to the framework:

1. Without crossing arcs, by construction, a semibracket always represents an arc whose other endpoint leans on the shortest structural arc covering the semibracket node. With crossing arcs, the other endpoint can be on a more external structural arc. If a semibracket is covered by several structural arcs and it encodes an arc leaning on the ith covering structural arc, we add an index $i-1$ to it (meaning that it will skip $i-1$ superbrackets when matching). An example can be seen in Figure 3a.

2. Structural arcs can cross other structural arcs. For this, we add an index i to a right superbracket (>,\) to indicate when it needs to skip i superbrackets on the stack (i.e., match the $(i+1)$th matching superbracket on the stack instead of the first), as in Figure 3b.

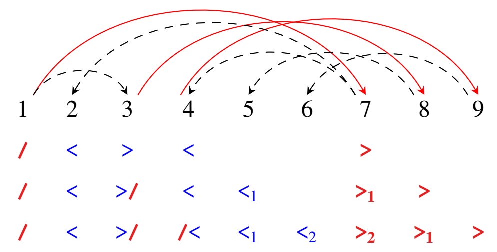

Figure 4: Step-by-step encoding of a non-projective tree. The bracketing sequence of the first row encodes the structural arc $(1 \to 7)$ with its auxiliary arcs $(1 \to 3)$, $(7 \to 2)$ and $(7 \to 4)$ The second row encodes $(3 \to 8)$ with $(8 \to 5)$, increasing the index of the 5th and 7th bracket to avoid decoding $(7 \to 5)$ or $(3 \to 7)$. The third row encodes $(4 \to 9)$ with $(9 \to 6)$, and increases twice the index of the 6th bracket, since $(6 \to 9)$ crosses with two structural arcs $(1 \to 7)$ and $(3 \to 8)$; and the 7th and 8th brackets, since they both cross with $(6 \to 9)$.

These extensions are sufficient to support every non-projective graph – note that auxiliary arcs can also cross other auxiliary arcs, as in Figure 3c, but no extension to labels is needed to support this: it is enough if the decoding algorithm ignores any intervening opening semibrackets on the stack when matching a closing semibracket (i.e., modifying lines 8 and 11 of Algorithm 1 to fetch the first superbracket instead of peeking).

Encoding is done step by step, by incrementally adding to the labels a structural arc and its associated auxiliary arcs at each step. Figure 4 shows a step-by-step example of this process.

To support the extension in the decoding algorithm, apart from the above change, it suffices to implement the following changes to Algorithm 1: (1) for closing semibrackets indexed with i, lines 8 and 11 fetch the $(i+1)$th opening superbracket from the stack instead of the first, (2) for closing superbrackets indexed with i, procedure CloseSuperbracket matches the $(i+1)$th opening superbracket instead of the first (behavior with intervening opening semibrackets is the same), and (3) for opening

	#trees	**#labels**					**#rels**		**#indices**			
		$\mathbf{B}_4$	$\mathbf{O}_\mathrm{P}$	$\mathbf{B}_7$	$\mathbf{O}_\mathrm{NP}$	$\mathbf{H}^+$	$\mathbf{O}_\mathrm{P}$	$\mathbf{O}_\mathrm{P}^+$	**0**	**1**	**2**	$\geq$ **3**
grc	13.9k	16	12	103	168	8	26	248	42.87	51.32	5.50	0.25
en	16.6	16	12	60	55	8	53	169	97.98	1.98	0.04	0
fi	15.1k	16	12	59	56	8	47	150	96.66	2.40	0.93	0
fr	16.3k	16	12	51	50	8	56	142	96.63	3.23	0.14	0
he	6.1k	16	12	37	30	8	37	52	99.22	0.75	0.03	0
ru	5k	16	12	60	52	8	42	153	95.27	4.31	0.42	0
ta	600	16	12	24	17	8	29	35	98.33	1.67	0	0
ug	3.5k	16	12	40	35	8	40	67	93.40	6.57	0.03	0
wo	2.1k	16	12	40	27	8	38	62	97.15	2.85	0	0
PTB	44k	16	12	22	21	8	45	46	99.90	0.10	0	0

Table 2: Treebank and encoding statistics. Number of trees in the dataset (**#trees**), generated labels (**#labels**) and dependency types (**#rels**), and % of trees that require indices $p \in \{0, 1, 2, \geq 3\}$ in non-projective labels. The notation is the following: $\mathbf{B}_4$ and $\mathbf{B}_7$ for the 4 and 7-bit encoding (Gómez-Rodríguez et al., 2023), respectively; $\mathbf{O}_\mathrm{P}$ (and $\mathbf{O}_\mathrm{NP}$) for the projective (and non-projective) optimal hierarchical bracketing; **H** for the hexatagging parser (Amini et al., 2023); and the symbol ($^+$) denotes if pseudo-projectivity is used.

semibrackets indexed with i, when matched (line 17) no arcs are created, instead they are placed back on the stack with their index decreased by one (and removed if it becomes zero). The resulting pseudocode is in Appendix A.1 (Algorithm 2), along with an example of an encoded sentence (Figure 6).

The bracket indexes are bounded by the maximum number of structural arcs that cover a fencepost between two nodes, minus one. For the optimal hierarchical bracketing, that maximum number (before subtracting one) is called *rope thickness* (Yli-Jyrä, 2019b), and coincides with the maximum number of open superbrackets that appear in the stack at the same time. Yli-Jyrä (2019b) presents shows that trees in UD 2.4 have rope thickness at most 8. This does not mean that we actually need brackets with index 7 to represent all trees in UD: rope thickness gives a conservative upper bound, as most of the time having k brackets in the stack does not mean that we need to skip to the deepest to encode the tree. In the set of treebanks used in our experiments, index 2 gives full coverage for every language except Ancient Greek.

5 Experiments

We conducted experiments in the Penn Treebank (Marcus et al., 1993, PTB) and 9 different languages from UD 2.14 (Nivre et al., 2020) to analyze the performance of our encodings in a diverse set of tree structures: Ancient Greek, English, Finnish, French, Hebrew, Russian, Tamil, Uyghur and Wolof. We report in this section the LAS and LCM metrics (Nivre et al., 2007) and in Appendix A.3 the detailed results. All our code is available at `github.com/anaezquerro/separ`.

We selected the biaffine parser (Dozat and Manning, 2017), and the hexatagging (Amini et al., 2023), 4-bit and 7-bit (Gómez-Rodríguez et al., 2023) encodings as baselines to assess the performance of our hierarchical encodings. For 4-bit and projective hierarchical bracketing, we also report results with the pseudo-projective transformation from Nivre and Nilsson (2005). In the case of hexatagging, we only report results with said transformation, following Amini et al. (2023).[4]

Model configuration We use XLM-RoBERTa (Conneau et al., 2020) as our encoder for all language treebanks, and XLNet (Yang et al., 2019) for English - both for our models and baselines - as it has shown some good results in previous work of parsing as tagging (Amini et al., 2023). For the decoder, we rely on two feed-forward networks for label and relation prediction.[5]

Results Table 2 shows the number of distinct labels generated by each encoding on each of treebank. For projective encodings, every treebank needs the 16 (for the 4-bit encoding) and 12 (for the optimal hierarchical bracketing) possible labels; whereas in the non-projective case the actual label usage varies per treebank. We see that the non-projective optimal hierarchical bracketing is more compact than the 7-bit encoding in every treebank except Ancient Greek (with an atypical amount of non-projectivity, since it contains poetry).

Accuracy results are in Table 3. The accuracy of our new optimal hierarchical bracketing encodings ($\mathbf{O}_\mathrm{P}$, $\mathbf{O}_\mathrm{NP}$) is roughly on par with the 4-bit, 7-bit and hexatagging baselines, outperforming them in some treebanks and falling slightly behind in others. $\mathbf{O}_\mathrm{P/NP}$ seem to do better in terms of LCM than LAS (e.g., $\mathbf{O}_\mathrm{P}$ outperforms the 4-bit encoding in a majority of treebanks in LCM, but not in LAS. Similar trends, albeit less pronounced, can be observed for the non-projective encodings). This makes sense as more compactness implies less redundancy in the encoding, so while getting a complete tree correct may be easier, a mistake can affect more arcs than in a more redundant encoding. We also tried, for the first time, pseudo-projectivity (Nivre and Nils-

[4]For homogeneous comparison, we use our own implementation of hexatagging, which we include as part of our code. Thus, there can be differences in hyperparameters with respect to the implementation in the original paper.

[5]See Appendix A.2 for more details.

	$\mathbf{B}_4$		$\mathbf{O}_{\mathrm{P}}$		$\mathbf{B}_4^+$		$\mathbf{O}_{\mathrm{P}}^+$		$\mathbf{B}_7$		$\mathbf{O}_{\mathrm{NP}}$		$\mathbf{H}^+$		**DM**	
grc	59.64	8.81	57.70	8.96	63.02	9.95	63.82	11.10	62.59	10.34	62.46	11.10	60.74	9.42	70.79	13.40
en	91.99	65.43	92.00	64.37	92.50	66.15	92.45	65.72	92.68	65.24	91.69	65.00	91.75	63.75	93.31	67.02
fi	88.04	42.38	88.13	44.31	89.32	45.98	88.65	44.74	89.11	45.27	88.58	45.27	88.65	44.76	81.87	0.00
fr	91.90	34.86	91.40	35.58	92.19	36.30	92.06	35.34	91.82	35.10	91.59	37.02	91.50	33.41	93.57	39.18
he	90.14	30.35	89.51	30.35	89.51	28.92	89.21	27.09	89.59	28.92	89.61	28.92	87.86	25.46	92.02	33.81
ru	87.99	31.11	87.82	30.62	88.09	28.79	88.18	30.95	88.40	30.95	87.06	28.79	88.71	30.62	90.60	36.44
ta	65.54	2.50	64.80	5.00	66.57	2.50	66.06	1.67	68.12	5.00	65.26	3.33	65.13	3.33	70.68	6.67
ug	66.62	11.67	66.27	11.89	66.78	11.22	66.88	12.33	66.47	10.33	65.17	11.22	65.56	11.00	71.39	14.56
wo	73.67	9.15	70.25	7.66	73.02	8.30	71.13	8.30	72.84	11.49	72.37	8.72	67.23	5.11	73.56	10.00
PTB	94.76	51.86	94.55	51.45	94.95	52.77	93.97	49.71	94.61	51.99	94.45	51.90	94.80	51.95	95.33	51.90
avg	81.03	28.81	80.24	29.02	81.60	29.09	81.24	28.70	81.62	29.46	80.82	29.13	80.19	27.88	83.31	27.30

Table 3: LAS and LCM (Labeled Complete Match) performance on the test sets. Same acronyms as in Table 2, and **DM** for the biaffine parser (Dozat and Manning, 2017). Best projective and non-projective bracketing encodings are underlined. Language abbreviations come from ISO 639-1.

son, 2005) on the 4-bit encoding and projective optimal hierarchical bracketing, which are projective encodings. The results show that it markedly boosts accuracy on non-projective treebanks, sometimes even beating native non-projective encodings.

If we compare hexatagging to all the bracketing-based encodings (including both the hierarchical encodings presented in this paper and the already-existing 4-bit and 7-bit), we can see that hexatagging excels in the PTB, but is worse on UD treebanks and especially suffers in Ancient Greek (the most non-projective treebank) and Wolof. On the other hand, the biaffine baseline obtains better LAS accuracy than every sequence-labeling encoding on most treebanks, but has the worst LCM on average.

Regarding efficiency, Figure 5 shows an efficiency comparison of the encodings of Table 3 on the English treebank, in terms of inference speed (tokens per second) vs. accuracy (LAS), highlighting the Pareto front. As can be seen, the projective and non-projective optimal hierarchical bracketing models are noticeably faster than the 4- and 7-bit baselines, likely due to their more compact label space. While hexatagging produces even fewer labels, it needs an intermediate conversion to a binary constituency tree and this penalizes runtime with respect to optimal hierarchical bracketing, which does not require it. All sequence labeling models are faster than the biaffine parser.

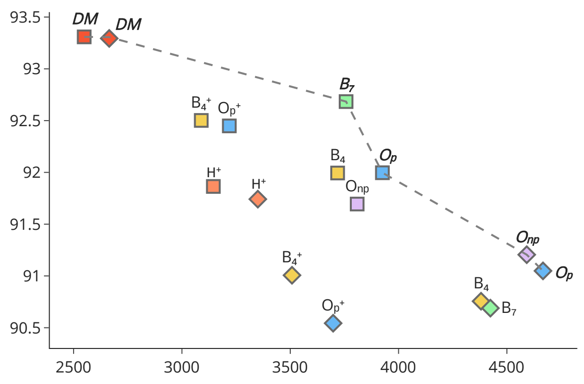

Figure 5: Performance (LAS, y-axis) vs inference speed (tokens per second, x-axis) on the English-EWT test set. The Pareto front is displayed with dashed lines and highlighted in bold. Same acronyms as in Table 3. Each decoder is displayed with a different color: ● **DM**, ● **H**$^+$, ● **B**$_4$, ● **B**$_7$, ● **O**$_{\mathrm{P}}$ and ● **O**$_{\mathrm{NP}}$; and each encoder with a different symbol: XLM ◇ and XLNet □.

6 Conclusion

We have advanced our understanding of bracketing encodings for dependency parsing with a framework that includes a spectrum of encodings from classic naive bracketing (all arcs are structural and require two brackets) to our novel optimal hierarchical bracketing (which minimizes the number of arcs that require two brackets), with the existing 4-bit encoding sitting in between. We have also defined and tested a new way of generalizing bracketing encodings to support non-projectivity, based on indexed brackets that skip other brackets when matching, showing that indexes bounded by 2 provide full coverage for most treebanks. Our experiments show that all our new encodings are competitive on a diverse set of treebanks.

From a wider perspective, our encodings are a step towards more compact modeling of the full search space of dependency trees. This includes non-projective trees, which are not supported by hexatagging (Amini et al., 2023); and potentially general graphs, which are naturally supported by our rope cover framework. Thus, an interesting avenue for future work is to apply optimal hierarchical bracketing to parse dependency graphs.

Acknowledgments

We acknowledge grants SCANNER-UDC (PID2020-113230RB-C21) funded by MICIU/AEI/10.13039/501100011033; GAP (PID2022-139308OA-I00) funded by MICIU/AEI/10.13039/501100011033/ and ERDF, EU; LATCHING (PID2023-147129OB-C21) funded by MICIU/AEI/10.13039/501100011033 and ERDF, EU; and TSI-100925-2023-1 funded by Ministry for Digital Transformation and Civil Service and "NextGenerationEU" PRTR; as well as funding by Xunta de Galicia (ED431C 2024/02), and CITIC, as a center accredited for excellence within the Galician University System and a member of the CIGUS Network, receives subsidies from the Department of Education, Science, Universities, and Vocational Training of the Xunta de Galicia. Additionally, it is co-financed by the EU through the FEDER Galicia 2021-27 operational program (Ref. ED431G 2023/01). We also extend our gratitude to CESGA, the supercomputing center of Galicia, for granting us access to its resources. Furthermore, we acknowledge the Faculty of Agriculture and Forestry of the University of Helsinki, as well as projects "Theory of Computational Logics" (352420) and "XAILOG" (345612, 345633) funded by the Research Council of Finland for the continued support of the third author during the multistage writing process.

Limitations

We adopt a default sequence labeling architecture. While the results are competitive and on par with the best-performing existing bracketing encodings, extensive fine-tuning could lead to further improvements. We refrain from doing so for two main reasons. First, our main goal is to demonstrate that the proposed encoding is effective and can be easily integrated with off-the-shelf methods without requiring extensive hyperparameter tuning. By doing so, we emphasize the practicality of our approach. Second, our computational resources are limited, consisting of six RTX 3090 GPUs, which are shared among multiple team members. Given these constraints, large-scale fine-tuning experiments would require a significant investment of time and resources, which is not feasible within our current setup. Instead, we prioritize demonstrating the effectiveness of our method under realistic conditions, where computational resources may be limited.

Ethical considerations

Our work does not raise ethical concerns. It presents a novel approach to dependency parsing using standard treebanks, addressing primarily computational and linguistic challenges. The applications and sources involved do not include sensitive personal data, human subjects, or scenarios likely to pose ethical issues. Therefore, the findings and techniques can be adopted within the field without ethical reservations.

We do, however, acknowledge the environmental impact of training neural models, particularly in terms of CO_2 emissions. All experiments were conducted in Spain, and we measured the carbon footprint per epoch during both training and inference. For our models, the emission was 1.44 g CO_2 (training) and 0.11 g CO_2 (inference). These values remain relatively low compared to those of recent large-scale NLP models, reinforcing the importance of pursuing energy-efficient approaches. For context, the European Union's emissions standard for newly manufactured cars is approximately 115 g CO_2 per kilometer.

References

Afra Amini, Tianyu Liu, and Ryan Cotterell. 2023. Hexatagging: Projective dependency parsing as tagging. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1453–1464, Toronto, Canada. Association for Computational Linguistics.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online. Association for Computational Linguistics.

Timothy Dozat and Christopher D. Manning. 2017. Deep Biaffine Attention for Neural Dependency Parsing. *Preprint*, arXiv:1611.01734.

Carlos Gómez-Rodríguez, Diego Roca, and David Vilares. 2023. 4 and 7-bit labeling for projective and non-projective dependency trees. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6375–6384, Singapore. Association for Computational Linguistics.

Carlos Gómez-Rodríguez, Michalina Strzyz, and David Vilares. 2020. A unifying theory of transition-based and sequence labeling parsing. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 3776–3793, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Ophélie Lacroix. 2019. Dependency parsing as sequence labeling with head-based encoding and multi-task learning. In *Proceedings of the Fifth International Conference on Dependency Linguistics (Depling, SyntaxFest 2019)*, pages 136–143.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. *Preprint*, arXiv:1711.05101.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Alberto Muñoz-Ortiz, Michalina Strzyz, and David Vilares. 2021. Not all linearizations are equally data-hungry in sequence labeling parsing. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021)*, pages 978–988, Held Online. INCOMA Ltd.

Joakim Nivre. 2006. Constraints on non-projective dependency parsing. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 73–80, Trento, Italy. Association for Computational Linguistics.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajič, Christopher D. Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers, and Daniel Zeman. 2020. Universal Dependencies v2: An evergrowing multilingual treebank collection. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 4034–4043, Marseille, France. European Language Resources Association.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 915–932, Prague, Czech Republic. Association for Computational Linguistics.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 99–106, Ann Arbor, Michigan. Association for Computational Linguistics.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2019. Viable dependency parsing as sequence labeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 717–723, Minneapolis, Minnesota. Association for Computational Linguistics.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2020. Bracketing encodings for 2-planar dependency parsing. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 2472–2484, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. 2015. Empirical Evaluation of Rectified Activations in Convolutional Network. *Preprint*, arXiv:1505.00853.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Anssi Yli-Jyrä. 2019a. How to embed noncrossing trees in universal dependencies treebanks in a low-complexity regular language. *Journal of Language Modelling*, 7(2):177–232.

Anssi Yli-Jyrä. 2019b. Transition-based coding and formal language theory for ordered digraphs. In *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*, pages 118–131, Dresden, Germany. Association for Computational Linguistics.

Anssi Yli-Jyrä and Carlos Gómez-Rodríguez. 2017. Generic axiomatization of families of noncrossing graphs in dependency parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1745–1755, Vancouver, Canada. Association for Computational Linguistics.

A Appendix

A.1 Non-projective decoding and example

Algorithm 2 shows the pseudocode of the non-projective extension for the optimal hierarchical bracketing encoding. In this case, since the modified algorithm accesses elements that are deep in the stack, the upper bound for runtime complexity is $O(|A|{i_{max}}^2)$, where $|A|$ is the number of arcs in the graph and i_{max} is the maximum bracket index, as an indexed left bracket may need to be updated at most i_{max} times to decrease its index while being at stack depth at most i_{max}. For trees, considering that setting $i_{max} = 2$ has full coverage on most treebanks, this becomes $O(n)$ in practice.

In Figure 6, we show an illustrative example of how this extension works on a graph with multiple crossing structural arcs.

A.2 Model configuration

Our neural models consist of a Transformer-based encoder module (XLM-RoBERTa$^{\mathrm{L}}$ or XLNet$^{\mathrm{L}}$ for

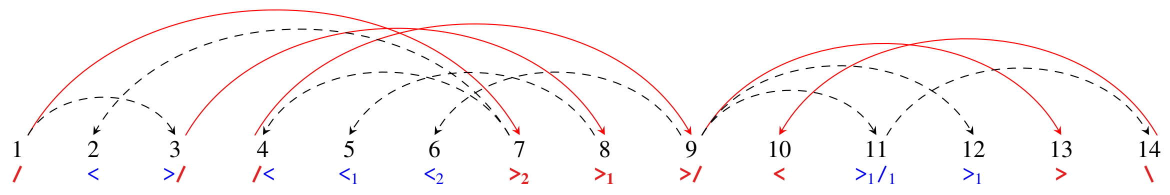

Figure 6: Non-projective extension for a complex graph.

Algorithm 2 Extension of Algorithm 1 for arbitrary graphs that can contain crossing arcs. The stack is no longer treated as a strict stack, since we need to be able to access the ith element to create crossing arcs. Thus, we assume the following functions: $\text{FETCH}(S, \textit{condition}, i)$ returns the ith element from the top of the stack (the top is the 1st) satisfying the condition, $\text{REMOVE}(S, \textit{condition}, i)$ does the same but removing the element, and $\text{PUT}(S, \textit{element}, i)$ inserts the given element at depth i from the top of the stack (with $i = 1$ to push to the top of the stack).

```
procedure DECODE($l_1, \ldots, l_n$)
    $S \leftarrow$ empty stack ; $A \leftarrow$ empty set
    for $i = 1 \rightarrow n$ do
        for symbol$_{index}$ in $l_i$ do                              ▷ index = 0 if the symbol has no index
            if symbol $\in \{<, /, <, /\}$ then
                PUSH($S$, (symbol$_{index}$, $i$))
            else if symbol $=$ $>$ then
                $(s_x, j) \leftarrow$ FETCH($S$, ISSUPERBRACKET, index+1)      ▷ $s \in \{/, <\}$
                $A \leftarrow A \cup \{j \rightarrow i\}$
            else if symbol $=$ \ then
                $(s_x, j) \leftarrow$ FETCH($S$, ISSUPERBRACKET, index+1)      ▷ $s \in \{/, <\}$
                $A \leftarrow A \cup \{j \leftarrow i\}$
            else if symbol $=$ $>$ or symbol $=$ \ then
                CLOSESUPERBRACKET(symbol$_{index}$, i, S, A)

procedure CLOSESUPERBRACKET(rbracket$_{rind}$, i, S, A)
    depth $\leftarrow 1$
    (lbracket$_{lind}$, $j$) $\leftarrow$ REMOVE($S$, TRUE, depth)
    while not ( (lbracket $=$ $<$ or lbracket $=$ $/$) and rind = 0 ) do
        if lbracket $=$ $<$ or lbracket $=$ $/$ then
            if lind > 0 then
                PUT($S$,(lbracket$_{lind-1}$, $j$), depth)
                depth $\leftarrow$ depth $+ 1$
            else if lbracket $=$ $<$ then $A \leftarrow A \cup \{j \leftarrow i\}$
            else $A \leftarrow A \cup \{j \rightarrow i\}$                          ▷ lbracket $=$ $/$
        else                                    ▷ lbracket $=$ $<$ or lbracket $=$ $/$, but rind $> 0$
            PUT($S$,(lbracket$_{lind}$, $j$), depth)
            depth $\leftarrow$ depth $+ 1$
        (lbracket$_{lind}$, $j$) $\leftarrow$ REMOVE($S$, TRUE, depth)
    if lbracket $=$ $<$ and rbracket $=$ \ then                                  ▷ rind $= 0$
        $A \leftarrow A \cup \{j \leftarrow i\}$
    else if lbracket $=$ $/$ and rbracket $=$ $>$ then
        $A \leftarrow A \cup \{j \rightarrow i\}$
```

English treebanks) stacked with two separate feed-forward networks that predict the sequence of labels and dependency relations using a multi-task cross-entropy loss. The encoder processes the input sentence and returns a contextualized embedding matrix, which is independently fed into each FFN to predict each output label at the token level. For both FFNs, we used the LeakyReLU activation (Xu et al., 2015) and optimized the full architecture with AdamW (Loshchilov and Hutter, 2019).

All models were trained for 100 epochs with a constant learning rate $\eta = 10^{-5}$, batch size of 300, and early stopping on the development set (in terms of UAS) with 20 epochs of patience.

A.3 Detailed results

Tables 5 to 13 present a breakdown of results per treebank. Each table includes the Performance on terms of unlabaled (UAS) and labeled (LAS) accuracy and unlabeled (UM) and labeled (LM) exact match on the development and test sets, along with theoretical and empirical coverage. We compute theoretical coverage based on the arcs recovered when applying the decoding process to the real labels. Empirical coverage is computed similarly but considers the labels and dependency relations that appear in the training set. Thus, empirical coverage sets the maximum performance that an encoding can achieve. Note that for the 4-bit and optimal hierarchical bracketing encodings, theoretical coverage is the same, as both cover only projective graphs and their respective pseudoprojective transformations.

		dev				test			
		UAS	**LAS**	**UM**	**LM**	**UAS**	**LAS**	**UM**	**LM**
th	B_4/O_P	100	100	100	100	100	100	100	100
	$B_4^+/O_P^+/H^+$	100	100	100	100	100	100	100	100
	B_7	100	100	100	100	100	100	100	100
	H^+	99.99	99.99	99.82	99.82	100	100	99.96	99.96
emp	B_4	100	100	100	100	100	100	100	100
	O_P	99.99	99.99	99.82	99.82	100	100	99.96	99.96
	B_4^+	100	100	100	100	100	100	100	100
	O_P^+	100	100	100	100	100	100	100	100
	B_7	100	100	100	100	100	100	100	100
	O_{NP}	100	100	100	100	100	100	100	100
XLNet	B_4	95.76	93.95	$\underline{62.24}$	47.59	96.32	$\underline{94.76}$	66.39	51.86
	O_P	95.58	93.79	61.29	47.29	96.07	94.55	65.23	51.45
	B_4^+	$\underline{95.94}$	$\underline{94.13}$	62.06	$\underline{48.35}$	$\underline{96.47}$	94.95	$\underline{66.47}$	$\underline{52.77}$
	O_P^+	94.92	93.28	59.88	47.59	95.52	93.97	63.45	49.71
	B_7	$\underline{95.67}$	93.76	$\underline{62.12}$	$\underline{47.12}$	$\underline{96.14}$	$\underline{94.61}$	$\underline{64.98}$	$\underline{51.99}$
	O_{NP}	95.59	$\underline{93.77}$	61.35	47.00	95.98	94.45	65.23	51.90
	H^+	95.67	93.83	62.18	47.53	96.30	04.80	66.43	51.95
	DM	96.31	94.44	62.71	48.12	96.84	95.33	65.60	51.90

Table 4: Performance on the PTB. Rows are grouped to display the theoretical (**th**) and empirical (**emp**) coverage and different encoders (**XLM**, **XLNet**).

		dev				test			
		UAS	**LAS**	**UM**	**LM**	**UAS**	**LAS**	**UM**	**LM**
th	B_4/O_P	99.80	99.80	98.50	98.50	99.83	99.83	98.94	98.94
	$B_4^+/O_P^+/H^+$	100	100	99.95	99.95	100	100	100	100
	B_7	100	100	100	100	100	100	100	100
emp	B_4	99.80	99.80	98.50	98.50	99.83	99.83	98.94	98.94
	O_P	99.84	99.84	98.35	98.35	99.88	99.88	98.89	98.89
	B_4^+	99.99	99.99	99.70	99.70	99.99	99.99	99.90	99.90
	O_P^+	99.99	99.99	99.70	99.70	99.99	99.99	99.90	99.90
	B_7	100	100	99.95	99.95	100	100	100	100
	O_{NP}	99.98	99.98	99.90	99.90	100	100	100	100
	H^+	99.99	99.99	99.70	99.70	99.99	99.99	99.90	99.90
XLM	B_4	93.51	91.27	71.31	60.42	93.01	90.76	71.83	61.53
	O_P	93.47	91.28	71.21	60.97	93.15	91.05	72.17	62.35
	B_4^+	93.76	91.62	70.76	60.72	93.27	91.01	72.36	62.16
	O_P^+	93.65	91.37	70.96	60.42	92.96	90.54	72.65	61.48
	B_7	93.64	91.39	70.81	60.02	92.99	90.69	72.22	61.92
	O_{NP}	93.89	91.62	71.56	60.32	93.59	91.21	72.56	61.96
	H^+	94.75	92.40	73.66	62.92	94.05	91.54	74.68	63.65
	DM	95.58	93.94	75.11	66.07	95.14	93.31	76.07	66.44
XLNet	B_4	94.77	93.01	74.11	64.92	94.14	91.99	74.82	65.43
	O_P	94.76	92.99	74.21	64.77	94.12	92.00	75.01	64.37
	B_4^+	$\underline{95.09}$	$\underline{93.28}$	$\underline{74.36}$	$\underline{65.02}$	$\underline{94.62}$	$\underline{92.50}$	75.69	$\underline{66.15}$
	O_P^+	94.72	92.87	74.26	64.27	94.51	92.45	$\underline{75.93}$	65.72
	B_7	$\underline{94.94}$	$\underline{93.05}$	74.66	64.57	$\underline{94.64}$	$\underline{92.68}$	$\underline{75.25}$	$\underline{65.24}$
	O_{NP}	94.69	92.86	$\underline{74.86}$	$\underline{64.57}$	93.87	91.69	74.29	65.00
	H^+	95.14	92.77	74.86	64.12	94.33	91.75	74.77	63.75
	DM	95.68	93.90	76.11	66.32	95.11	93.31	77.27	67.02

Table 5: Performance on the English EWT.

		dev				test			
		UAS	**LAS**	**UM**	**LM**	**UAS**	**LAS**	**UM**	**LM**
th	B_4/O_P	86.16	86.16	40.28	40.28	85.44	85.44	47.32	47.32
	$B_4^+/O_P^+/H^+$	99.44	99.44	90.33	90.33	99.44	99.44	92.5	92.5
	B_7	99.33	99.33	95.34	95.34	99.24	99.24	95.25	95.25
emp	B_4	86.16	86.16	40.28	40.28	85.44	85.43	47.32	47.24
	O_P	87.80	87.80	28.76	28.76	88.16	88.15	38.67	38.59
	B_4^+	99.03	99.03	85.31	85.31	99.23	99.22	90.20	89.97
	O_4^+	99.03	99.03	85.31	85.31	99.23	99.22	90.20	89.97
	B_7	99.29	99.29	95.25	95.25	99.2	99.19	95.1	94.87
	O_{NP}	99.83	99.83	99.21	99.21	99.94	99.92	99.69	99.46
	H^+	98.99	98.99	85.66	85.66	99.14	99.13	89.66	89.43
XLM	B_4	67.19	60.04	10.55	5.36	66.74	59.64	14.70	8.81
	O_P	65.37	58.42	7.56	3.87	64.52	57.70	13.78	8.96
	B_4^+	70.67	63.08	10.99	6.42	70.83	63.02	16.54	9.95
	O_P^+	$\underline{70.90}$	$\underline{63.58}$	$\underline{12.75}$	$\underline{7.04}$	$\underline{71.24}$	$\underline{63.82}$	$\underline{18.15}$	$\underline{11.10}$
	B_7	$\underline{70.25}$	$\underline{62.61}$	11.52	6.51	$\underline{70.39}$	$\underline{62.59}$	17.38	10.34
	O_{NP}	69.37	62.57	$\underline{12.84}$	$\underline{8.00}$	69.51	62.46	$\underline{17.53}$	$\underline{11.10}$
	H^+	67.98	60.57	10.82	6.68	68.74	60.74	16.00	9.42
	DM	77.37	70.25	15.92	9.32	78.36	70.79	22.89	13.40

Table 6: Performance on the Ancient-Greek Perseus.

		dev				test			
		UAS	LAS	UM	LM	UAS	LAS	UM	LM
th	B_4/O_P	99.23	99.23	95.01	95.01	99.18	99.18	95.56	95.56
	$B_4^\ddagger/O_P^\ddagger/H^+$	100	100	99.93	99.93	100	100	99.94	99.94
	B_7	99.99	99.99	99.93	99.93	100	100	99.94	99.94
emp	B_4	99.23	99.23	95.01	95.01	99.18	99.18	95.56	95.56
	O_P	98.76	98.76	93.18	93.18	98.81	98.81	94.08	94.08
	$B_4^\ddagger$	99.97	99.97	99.34	99.34	99.99	99.99	99.74	99.74
	$O_P^\ddagger$	99.97	99.97	99.34	99.34	99.99	99.99	99.74	99.74
	B_7	99.99	99.99	99.93	99.93	99.99	99.99	99.87	99.87
	O_{NP}	100	100	100	100	99.96	99.96	99.87	99.87
	H^+	99.97	99.97	99.34	99.34	99.99	99.99	99.74	99.74
XLM	B_4	90.93	87.52	57.40	42.74	91.49	88.04	58.33	42.38
	O_P	91.40	87.96	60.12	45.01	91.29	88.13	59.49	44.31
	$B_4^\ddagger$	92.71	89.48	63.56	47.51	92.58	89.32	61.48	45.98
	$O_P^\ddagger$	92.16	88.83	61.29	46.11	92.00	88.65	60.00	44.37
	B_7	92.33	89.09	62.68	47.14	92.38	89.11	61.67	45.27
	O_{NP}	91.80	88.59	60.85	45.89	91.85	88.58	61.35	45.27
	H^+	92.17	88.68	62.46	46.04	92.16	88.65	60.90	44.76
	DM	94.42	81.00	67.52	0.00	94.94	81.87	68.42	0.00

Table 7: Performance on the Finnish TDT.

		dev				test			
		UAS	LAS	UM	LM	UAS	LAS	UM	LM
th	B_4/O_P	99.98	99.98	99.79	99.79	100	100	100	100
	$B_4^\ddagger/O_P^\ddagger/H^+$	100	100	100	100	100	100	100	100
	B_7	100	100	100	100	100	100	100	100
emp	B_4	99.98	99.98	99.79	99.79	100	100	100	100
	O_P	99.95	99.95	99.38	99.38	99.90	99.90	99.59	99.59
	$B_4^\ddagger$	100	100	100	100	100	100	100	100
	$O_4^\ddagger$	100	100	100	100	100	100	100	100
	B_7	100	100	100	100	100	100	100	100
	H^+	100	100	100	100	100	100	100	100
	O_{NP}	100	100	100	100	100	100	100	100
XLM	B_4	93.83	91.02	48.14	35.33	92.78	90.14	42.77	30.35
	O_P	93.47	90.39	46.49	32.85	92.06	89.51	42.57	30.35
	$B_4^\ddagger$	93.64	90.86	47.11	33.47	92.21	89.51	40.33	28.92
	$O_4^\ddagger$	93.43	90.47	48.35	34.09	91.79	89.21	40.33	27.09
	B_7	93.59	91.03	46.69	36.16	92.33	89.59	42.77	28.92
	O_{NP}	93.88	90.39	50.21	33.88	92.32	89.61	42.16	28.92
	H^+	92.37	88.64	45.04	27.89	91.25	87.86	38.70	25.46
	DM	95.87	93.42	54.55	40.70	94.42	92.02	48.47	33.81

Table 9: Performance on the Hebrew HTB.

		dev				test			
		UAS	LAS	UM	LM	UAS	LAS	UM	LM
th	B_4/O_P	99.46	99.46	96.27	96.27	99.35	99.35	95.91	95.91
	$B_4^\ddagger/O_P^\ddagger/H^+$	99.98	99.98	99.59	99.59	99.99	99.99	99.76	99.76
	B_7	100	100	100	100	100	100	100	100
emp	B_4	99.46	99.46	96.27	96.27	99.35	99.35	95.91	95.91
	O_P	99.69	99.69	96.41	96.41	99.67	99.67	96.15	96.15
	$B_4^\ddagger$	99.98	99.98	99.46	99.46	99.98	99.98	99.28	99.28
	$O_P^\ddagger$	99.98	99.98	99.46	99.46	99.98	99.98	99.28	99.28
	B_7	100	100	100	100	100	100	100	100
	O_{NP}	100	100	100	100	100	100	100	100
	H^+	99.98	99.98	99.46	99.46	99.98	99.98	99.28	99.28
XLM	B_4	96.19	94.09	58.74	43.36	94.65	91.90	50.24	34.86
	O_4	96.13	94.08	58.67	42.82	94.33	91.40	51.68	35.58
	$B_4^\ddagger$	96.52	94.42	60.09	44.17	94.70	92.19	49.28	36.30
	$O_P^\ddagger$	96.44	94.39	59.82	44.38	94.81	92.06	49.04	35.34
	B_7	96.46	94.42	59.55	44.58	94.45	91.82	48.32	35.10
	O_{NP}	96.37	94.23	59.96	44.04	94.14	91.59	48.80	37.02
	H^+	95.78	93.75	58.13	43.29	94.53	91.50	50.48	33.41
	DM	97.33	95.46	62.87	46.75	96.03	93.57	52.16	39.18

Table 8: Performance on the French GSD.

		dev				test			
		UAS	LAS	UM	LM	UAS	LAS	UM	LM
th	B_4/O_P	99.41	99.41	95.16	95.16	99.44	99.44	95.84	95.84
	$B_4^\ddagger/O_P^\ddagger/H^+$	99.98	99.98	99.31	99.31	99.96	99.96	99.17	99.17
	B_7	100	100	100	100	100	100	100	100
emp	B_4	99.41	99.41	95.16	95.16	99.44	99.43	95.84	95.84
	O_P	99.39	99.39	94.47	94.47	99.23	99.23	94.34	94.34
	$B_4^\ddagger$	99.90	99.90	97.41	97.41	99.81	99.81	96.51	96.51
	$O_P^\ddagger$	99.90	99.90	97.41	97.41	99.81	99.81	96.51	96.51
	B_7	99.97	99.97	99.83	99.83	100	100	100	99.83
	O_{NP}	99.99	99.99	99.65	99.65	99.98	99.98	99.83	99.67
	H^+	99.90	99.90	97.41	97.41	99.81	99.81	96.51	96.51
XLM	B_4	91.83	88.95	44.39	32.99	91.76	87.99	45.42	31.11
	O_P	91.15	88.17	41.28	30.92	91.49	87.82	46.92	30.62
	$B_4^\ddagger$	92.12	89.27	41.97	31.61	92.08	88.09	45.59	28.79
	$O_P^\ddagger$	91.29	88.58	43.01	32.30	91.85	88.18	46.92	30.95
	B_7	91.66	88.73	41.45	30.40	92.14	88.40	46.26	30.95
	O_{NP}	90.85	88.03	41.11	30.40	90.85	87.06	44.76	28.79
	H^+	90.13	87.45	40.07	30.92	91.63	87.61	45.59	30.62
	DM	93.94	91.50	47.67	37.48	93.99	90.60	53.74	36.44

Table 10: Performance on the Russian GSD.

		dev				test			
		UAS	**LAS**	**UM**	**LM**	**UAS**	**LAS**	**UM**	**LM**
th	B_4/O_P	100	100	100	100	99.87	99.87	98.33	98.33
	$B_4^+/O_P^+/H^+$	100	100	100	100	100	100	100	100
	B_7	100	100	100	100	100	100	100	100
emp	B_4	100	99.96	100	98.75	99.87	99.87	98.33	98.33
	O_P	100	99.96	100	98.75	99.82	99.82	97.50	97.50
	B_4^+	100	99.96	100	98.75	99.84	99.84	97.50	97.50
	O_P^+	100	99.96	100	98.75	99.84	99.84	97.50	97.50
	B_7	100	99.96	100	98.75	99.87	99.87	98.33	98.33
	O_{NP}	100	99.96	100	98.75	99.93	99.93	99.17	99.17
	H^+	100	99.96	100	98.75	99.84	99.84	97.50	97.50
XLM	B_4	78.72	69.10	17.50	6.25	76.07	65.54	14.17	2.50
	O_P	79.30	69.03	$\underline{23.75}$	8.75	75.82	64.80	$\underline{16.67}$	$\underline{5.00}$
	B_4^+	$\underline{79.33}$	69.64	20.00	6.25	$\underline{77.11}$	$\underline{66.57}$	15.83	2.50
	O_P^+	78.52	$\underline{69.73}$	16.25	$\underline{10.00}$	76.69	66.06	15.83	1.67
	B_7	$\underline{79.06}$	$\underline{68.97}$	$\underline{18.75}$	$\underline{8.75}$	$\underline{78.02}$	$\underline{68.12}$	$\underline{19.17}$	$\underline{5.00}$
	O_{NP}	77.76	67.37	15.00	5.00	77.09	65.26	19.17	3.33
	H^+	78.22	67.23	16.25	5.00	76.26	65.13	17.50	3.33
	DM	82.85	73.47	25.00	10.00	81.48	70.68	24.17	6.67

Table 11: Performance on the Tamil TTB.

		dev				test			
		UAS	**LAS**	**UM**	**LM**	**UAS**	**LAS**	**UM**	**LM**
th	B_4/O_P	99.01	99.01	92.33	92.33	98.66	98.66	92.33	92.33
	$B_4^+/O_P^+/H^+$	99.98	99.98	99.44	99.44	99.96	99.96	99.33	99.33
	B_7	100	100	100	100	100	100	100	100
emp	B_4	98.83	98.75	93.11	92.11	98.50	97.96	93.44	87.44
	O_P	99.01	98.94	92.33	91.33	98.66	98.13	92.33	86.44
	B_4^+	99.83	99.76	97.67	96.56	99.81	99.25	97.33	91.22
	O_P^+	99.83	99.76	97.67	96.56	99.81	99.25	97.33	91.22
	B_7	98.83	98.75	93.11	92.11	98.50	97.96	93.44	87.44
	O_{NP}	99.88	99.81	99.22	98.00	99.68	99.13	99.00	92.78
	H^+	99.82	99.75	97.44	96.33	99.8	99.25	97.33	91.22
XLM	B_4	81.50	$\underline{68.22}$	$\underline{31.67}$	$\underline{11.44}$	$\underline{80.03}$	66.62	$\underline{30.00}$	11.67
	O_P	80.26	67.02	29.89	10.67	79.37	66.27	29.44	11.89
	B_4^+	$\underline{81.50}$	68.10	30.78	10.78	79.70	66.78	29.00	11.22
	O_P^+	80.83	67.78	30.22	10.33	79.55	$\underline{66.88}$	28.78	$\underline{12.33}$
	B_7	$\underline{81.03}$	$\underline{67.49}$	30.67	$\underline{10.89}$	$\underline{80.09}$	$\underline{66.47}$	$\underline{30.11}$	10.33
	O_{NP}	80.15	66.23	$\underline{30.78}$	9.11	78.70	65.17	29.33	$\underline{11.22}$
	H^+	81.65	67.37	32.33	10.11	79.52	65.56	29.33	10.00
	DM	85.08	72.84	38.44	15.22	84.16	71.39	38.44	14.56

Table 12: Performance on the Uyghur UDT.

		dev				test			
		UAS	**LAS**	**UM**	**LM**	**UAS**	**LAS**	**UM**	**LM**
th	B_4/O_P	99.68	99.68	96.44	96.44	99.64	99.64	97.02	97.02
	$B_4^+/O_P^+/H^+$	100	100	100	100	99.99	99.99	99.79	99.79
	B_7	100	100	100	100	100	100	100	100
emp	B_4	99.71	99.69	96.88	96.44	99.58	99.57	96.81	96.60
	O_P	99.68	99.67	96.44	95.99	99.64	99.63	97.02	96.81
	B_4^+	99.93	99.92	98.22	97.77	99.93	99.92	98.72	98.51
	O_P^+	99.93	99.92	98.22	97.77	99.93	99.92	98.72	98.51
	B_7	99.71	99.69	96.88	96.44	99.58	99.57	96.81	96.60
	O_{NP}	99.92	99.91	99.55	99.11	100	99.99	100	99.79
	H^+	99.93	99.92	98.0	97.55	99.90	99.89	98.3	98.09
XLM	B_4	$\underline{81.15}$	$\underline{73.82}$	$\underline{15.37}$	$\underline{6.90}$	$\underline{80.82}$	$\underline{73.67}$	18.30	$\underline{9.15}$
	O_P	79.30	71.03	14.03	6.46	78.33	70.25	18.09	7.66
	B_4^+	80.43	72.78	13.81	5.12	80.37	73.02	$\underline{19.15}$	8.30
	O_P^+	79.35	71.46	13.36	5.79	78.81	71.13	18.72	8.30
	B_7	$\underline{80.54}$	$\underline{72.94}$	15.14	$\underline{6.68}$	$\underline{80.62}$	$\underline{72.84}$	$\underline{21.28}$	$\underline{11.49}$
	O_{NP}	80.47	72.58	$\underline{16.70}$	6.68	79.90	72.37	18.94	8.72
	H^+	77.08	68.13	13.14	4.68	75.65	67.23	15.11	5.11
	DM	80.70	72.37	16.70	6.90	81.34	73.56	20.43	10.00

Table 13: Performance on the Wolof WTB.