Køpsala
Transition-Based Graph Parsing via Efficient Training and Effective Encoding
Hershcovich, Daniel; De Lhoneux, Miryam; Kulmizev, Artur; Pejhan, Elham; Nivre, Joakim

Published in:
Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies

DOI:
10.18653/v1/2020.iwpt-1.25

Publication date:
2020

Document version
Publisher's PDF, also known as Version of record

Document license:
CC BY

Citation for published version (APA):
Køpsala: Transition-Based Graph Parsing via Efficient Training and Effective Encoding

Daniel Hershcovich♦ Miryam de Lhoneux♦ Artur Kulmizev♥
Elham Pejhan♦ Joakim Nivre♥

♦University of Copenhagen ♥Uppsala University

{dh,ml,ep}@di.ku.dk,
{artur.kulmizev,joakim.nivre}@lingfil.uu.se

Abstract

We present Køpsala, the Copenhagen-Uppsala system for the Enhanced Universal Dependencies Shared Task at IWPT 2020. Our system is a pipeline consisting of off-the-shelf models for everything but enhanced graph parsing, and for the latter, a transition-based graph parser adapted from Che et al. (2019). We train a single enhanced parser model per language, using gold sentence splitting and tokenization for training, and rely only on tokenized surface forms and multilingual BERT for encoding. While a bug introduced just before submission resulted in a severe drop in precision, its post-submission fix would bring us to 4th place in the official ranking, according to average ELAS. Our parser demonstrates that a unified pipeline is effective for both Meaning Representation Parsing and Enhanced Universal Dependencies.

1 Introduction

The IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies (Bouma et al., 2020) involves sentence segmentation, tokenization, lemmatization, part-of-speech tagging, morphological analysis, basic dependency parsing, and finally (for the first time) enhanced dependency parsing. The enhancements encode case information, elided predicates, and shared arguments due to conjunction, control, raising and relative clauses (see Figures 1 and 2).

In Universal Dependencies v2 (UD; Nivre et al., 2020), enhanced dependencies (ED) are a separate dependency graph than the basic dependency tree (BD). However, ED is almost a super-set of BD, and so most previous approaches (Schuster and Manning, 2016; Nivre et al., 2018) have attempted to recover ED from BD using language-specific rules. On the other hand, Hershcovich et al. (2018) experimented with TUPA, a transition-based directed acyclic graph (DAG) parser originally designed for parsing UCCA (Abend and Rappoport, 2013), for supervised ED parsing. They converted ED to UCCA-like graphs and did not use pre-trained contextualized embeddings, yielding sub-optimal results. Taking a similar approach, we adapt a transition-based graph parser (Che et al., 2019) designed for Meaning Representation Parsing (Oepen et al., 2019), but parse ED directly and use BERT embeddings (Devlin et al., 2019).

Figure 1: ED for reviews-077034-0002 from UD_English-EWT, containing a control verb (made). Arrows above the sentence are also in BD.

The main contribution of our work is a transition system supporting the graph structures exhibited by ED, including null nodes (meaning this is not a strictly bilexical formalism), cycles and non-crossing graphs (§3.1), as Figure 4 demonstrates for the sentence from Figure 2. We parse ED completely separately from BD, demonstrating the applicability of a full graph parser, starting from only segmented and tokenized text, to ED. Our code is available at https://github.com/coastalcph/koepsala-parser.

2 Preprocessing

As the focus of this shared task is ED parsing, we rely on existing systems for preprocessing. Here, we consider two off-the-shelf pipelines: STANZA

---

*Equal contribution

1Some BD arcs are deleted in ED, e.g., orphan arcs.
Deze is de modernste en grootste hal van België, en de enige die voldoet aan de Olympische normen.

(Qi et al., 2020)\(^2\) and udpipe 1.2 (Straka and Straková, 2017; Straka et al., 2016),\(^3\) both of which have models pre-trained on UD v2.5 treebanks. We experiment with either pipeline during prediction to process the raw text files for the dev and test sets, eventually selecting UDPipe for our primary submission. This process entails sentence segmentation, tokenization, lemmatization, part-of-speech tagging, morphological feature tagging, and BD parsing.\(^4\) For training our ED parser (§3), however, we use gold inputs for simplicity. We use the conllu Python package\(^5\) to read CoNLL-U files.

Preprocessing model selection. Since the dev and test data do not denote their source treebanks, we simply process the text using the pipeline model trained on the language’s largest treebank. To experiment with an alternative method, for languages with more than one treebank, we also train UDPipe models on combined training treebanks. Table 1 shows the comparison of LAS on the combined dev set, for these models and for the models (pre-)trained on the language’s largest treebank. The results show that using the combined training sets does not lead to consistent improvements in terms of LAS, and so we continue using pre-trained treebank-specific preprocessing models henceforth.

### Table 1: LAS on the combined dev set for UDPipe models trained on the language’s combined training treebanks and the models trained on the language’s largest treebank. No consistent trend is observed.

<table>
<thead>
<tr>
<th>Language</th>
<th>Czech</th>
<th>Dutch</th>
<th>Estonian</th>
<th>Polish</th>
</tr>
</thead>
<tbody>
<tr>
<td>combined</td>
<td>78.88</td>
<td>76.50</td>
<td>77.01</td>
<td>82.96</td>
</tr>
<tr>
<td>largest</td>
<td>83.97</td>
<td>74.97</td>
<td>77.61</td>
<td>82.59</td>
</tr>
</tbody>
</table>

(2) https://stanfordnlp.github.io/stanza/models.html
(3) https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-3131
(4) The preprocessing output, except for segmentation and tokenization, is not used in any way by the ED parser, since it just uses BERT for token representation (§3.2).
(5) https://github.com/EmilStenstrom/conllu

---

Figure 2: wiki-3745.p.38.s.5 from UD_Dutch-LassySmall, containing a null node NULL, not in the original sentence, coordination and case suffixes (:en :van :aan), and propagation of conjuncts (hal → grootste). The dashed edges are deleted in ED, and the edges below the sentence are added. Note the cycle NULL ↔ voldoet.

3 Transition-Based Enhanced Dependency Parser

Our system is a transition-based graph parser, based on the HIT-SCIR system (Che et al., 2019), which achieved the highest average score across frameworks (AMR, EDS, UCCA, DM and PSD) in the CoNLL 2019 shared task on Meaning Representation Parsing (MRP; Oepen et al., 2019). It is written in the AllenNLP (Gardner et al., 2018) framework. For training efficiently, it employs stack LSTMs (Dyer et al., 2015), batching operations across sentences. For better encoding, HIT-SCIR fine-tuned BERT (Devlin et al., 2019) while training the parser.

A transition-based parser operates by manipulating a buffer (originally containing the input words provided by the preprocessor, see §2) and a stack (originally containing the root, i.e., word at index 0), to incrementally create the output dependency graph. At each point in the parsing process, a tran-
sion is selected out of a pre-defined set of possible transitions. A classifier is trained to predict the best transition to apply at each step, by mimicking an oracle during training (see §3.1).

HIT-SCIR used a different transition system per framework (AMR, EDS, UCCA; and one system for DM and PSD), according to the graph properties of each and based on existing framework-specific parsers (Liu et al., 2018; Buys and Blunsom, 2017; Hershcovich et al., 2017; Wang et al., 2018). We construct a transition system for ED using subsets of transitions from two of the HIT-SCIR systems: their system for DM and PSD, as well as their system for UCCA, with some further adaptations specific to ED graphs.

3.1 Transition System

Our system contains the following transitions: \{\textsc{shift}, \textsc{left-edge}$_i$, \textsc{right-edge}$_j$, \textsc{reduce-0}, \textsc{reduce-1}, \textsc{node}, \textsc{swap} and \textsc{finish}\}. The \textsc{shift} transition pops the first element of the buffer and pushes it onto the stack. The \textsc{left-edge}$_i$ and \textsc{right-edge}$_j$ transitions add an arc\(^6\) between the two top items of the stack with label \(l\). We need two different \textsc{reduce} transitions to pop the topmost and second topmost items of the stack, which we name \textsc{reduce-0} and a \textsc{reduce-1} respectively. This makes it possible to construct length-2 cycles, which ED allows (and most MRP frameworks do not). The \textsc{node} transition inserts a null node as the first element of the buffer, needed to support null nodes. \textsc{swap} moves the second-top node of the stack to the buffer, thus swapping the order between the two top nodes of the stack. This is necessary for handling crossing graphs (analogous to non-projective trees). Finally, \textsc{finish} terminates the transition sequence. A formal definition of the transition set is shown in Figure 3.

Separate \textsc{edge} transitions exist for each edge label. Labels containing coordination or case suffixes (such as \textsc{nmod:van}) are treated as any other label and are not split, resulting in a large number of transitions for some languages, shown in Table 2.

\textsc{node} transitions, on the other hand, do not se-

\begin{table}[h]
\centering
\begin{tabular}{|c|ccc|}
\hline
Language & Total & \textsc{edge} & w/ suffix \\
\hline
\textsc{arabic} & 402 & 395 & 345 \\
\textsc{bulgarian} & 197 & 191 & 137 \\
\textsc{czech} & 768 & 761 & 702 \\
\textsc{dutch} & 393 & 386 & 336 \\
\textsc{english} & 300 & 293 & 232 \\
\textsc{estonian} & 445 & 438 & 381 \\
\textsc{finnish} & 266 & 259 & 210 \\
\textsc{french} & 112 & 106 & 59 \\
\textsc{italian} & 281 & 274 & 216 \\
\textsc{latvian} & 238 & 232 & 161 \\
\textsc{lithuanian} & 323 & 317 & 263 \\
\textsc{polish} & 676 & 669 & 615 \\
\textsc{russian} & 944 & 938 & 861 \\
\textsc{slovak} & 266 & 259 & 204 \\
\textsc{swedish} & 209 & 202 & 153 \\
\textsc{tamil} & 146 & 140 & 103 \\
\textsc{ukrainian} & 290 & 283 & 225 \\
\hline
\end{tabular}
\caption{Number of transitions for each language.}
\end{table}

\(^6\)For consistency, we keep the transition nomenclature using “\textsc{edge}”, although they create directed dependency \textit{arcs}. Note that in analogous transitions in some transition systems, such as \textsc{arc-eager} (Nivre, 2003), the dependent of the transition is also popped out of the stack as part of either of these two transitions. Here, since dependents can have multiple heads and can have arcs with multiple labels, we stick to the \textsc{edge} action and use our two \textsc{reduce} transitions to pop elements of the stack when necessary.

\textbf{Constraints.} In addition to the modified transition set, we change the constraints for some transitions according to the required graph structure.

Since \textsc{left-edge}$_i$ and \textsc{right-edge}$_j$ transitions do not reduce the dependent, we need to ensure that we do not draw the same arc twice. For this reason, these transitions are not allowed if there is already an arc with label \(l\) between the two nodes. We also disallow to add an arc with the root as dependent.

To ensure every node gets attached to at least one head, we disallow the \textsc{reduce-0} and \textsc{reduce-1} transitions for nodes that do not have a head yet. We also disallow reducing the root.

For the \textsc{swap} transition, we maintain the \textit{generated order} of each node, assigned when the node is shifted (for words) or created (for null nodes). To prevent infinite loops during inference, we only allow swapping nodes whose order in the stack is the same as their generation order.

To limit repeated actions, we arbitrarily constrain \textsc{node} transitions such that there are no more null nodes than words (although a lower limit would suffice), and \textsc{edge} transitions to limit the
number of heads per node to 7.\(^7\)

Finish is only allowed when the buffer is empty and the stack only contains the root. If no valid transition is available, the sequence is terminated prematurely by applying the Finish transition, regardless of the Finish constraints.

Oracle. We use a static oracle similar to HIT-SCIR (a single “gold” transition sequence is given during training, which the parser is forced to follow), but develop one for our transition system.

The oracle deterministically chooses the transition to take given the current configuration. Let \( s_1 \) and \( s_0 \) be the two top items of the stack and \( b \) the first item of the buffer (if these are defined in the current configuration). If the buffer is empty and the stack only contains the root, take a Finish transition. Otherwise, if there is an arc between \( s_1 \) and \( s_0 \) with label \( l \) that has not yet been constructed, take the necessary Right-Edge\(_l\) or Left-Edge\(_l\) action. Otherwise, if \( s_0 \) has a node dependent, take a Node transition. Otherwise, if \( s_0 \) has all its heads and dependents, take Reduce-0, if \( s_1 \) has all its heads and dependents, take Reduce-1. Otherwise, if \( s_1 \) and \( s_0 \) are in their generated order and \( s_0 \) has a head or a dependent in the stack that is not \( s_1 \), take a Swap. Otherwise Shift. Figure 4 shows an example transition sequence.

3.2 Classifier

The parser uses BERT (Devlin et al., 2019) for token representation. While Che et al. (2019) used pre-trained English model (wwm-cased_L-24_H-102416), we replaced it with a pre-trained multilingual one (multi_cased_L-12_H-76812).\(^8\) trained on 104 languages, including all 17 languages participating in the shared task. As done by Che et al. (2019), we use the bert-pretrained text field embedder from AllenNLP, which extracts the first word-piece of each token, applying a scalar mix on all layers of transformer.

The transition classifier is a stack-LSTM (Dyer et al., 2015) with only BERT embedding features for words, as well as a scalar feature denoting the ratio between the number of (null) nodes and the number of words (Hershcovich et al., 2017), as in HIT-SCIR. We do not fine-tune BERT due to memory limitations, though fine-tuning would likely result in improved performance.

3.3 Postprocessing

The enhanced graphs are required to be connected, i.e., every node must be reachable from the root.\(^9\) While the transition constraints ensure that every node has a head, there may be unconnected cycles at the end of the parse, resulting in invalid graphs. To fix the problem, at the end of the parse, we iteratively find the unconnected node with the most descendants, and attach it to the predicate (the first dependent of the root) with an orphan-labeled arc. In addition to unconnected cycles, this resolves the problem of prematurely terminated transition sequences due to no valid transition being available according to the constraints: instead of resulting in partially-constructed graphs, headless nodes are similarly attached with an orphan-labeled arc to the predicate, if it exists, or otherwise to the root.

Parsing tragedy. Our postprocessing procedure to attach unconnected subgraphs had a bug at the time of submission, where many nodes were incorrectly identified as unconnected and thus un-

\(^7\)While the observed number of heads per node in the data goes up to 36, in the training data there is only a small minority of cases where a node has more than 7 heads.

\(^8\)https://github.com/google-research/bert/blob/master/multilingual.md

\(^9\)This is enforced by the task organizers by running validate.py --level 2 --lang ud on the system predictions before evaluation.
necessarily attached to the predicate/root. While this still yielded valid graphs, precision dropped precipitously from before the introduction of the postprocessing procedure. Due to the late stage in the evaluation period at which we made this change, we failed to properly monitor our development scores and could not identify the cause for the drop in time, resulting in low official scores. However, after submission we identified the bug and fixed it, improving our parser’s accuracy back to the range we had observed during development.

### 3.4 Training

For training the ED parser we do not simply train it on the largest treebank per language, but rather train it on the concatenated training treebanks per language. In preliminary experiments, this did lead to improvements in terms of combined dev ELAS over treebank-specific models, contrary to our findings in BD parsing for preprocessing (§2). We train our models on an NVIDIA P100 GPU with a batch size of 8. All other hyperparameters can be found in the configuration files in the repository.11

Training until convergence took 1h30 (for Tamil, 10https://github.com/coastalcph/koepsala-parser/blob/master/config/transition_eud.jsonnet

---

**Figure 4:** Oracle transition sequence for the sentence from Figure 2. Consecutive Shifts grouped for brevity.
the smallest treebank) to up to 2 days (for Arabic, which contains many long sentences). Prediction on the dev set took between 4 minutes (for Tamil) and 55 minutes (for Czech), ranging from 117 words/second (7 sentences/second, for Tamil) to 1300 words/second (81 sentences/second, for Czech), including the model loading time.

3.5 Baselines

In addition to providing validation scores for our trained parsers, we consider three competitive baselines, as provided by the task organizers:

- **B1**: gold standard dependency trees copied as enhanced graphs. Though this can be technically considered an upper bound, as gold tree information is provided, it should nonetheless provide some idea of how much of the enhanced graph can be derived from the dependency tree.

- **B2**: predicted trees yielded by UDPipe models trained on UD v2.5 (using the largest treebank where applicable), copied as enhanced graphs. This is more representative than **B1** of realistic parsing scenarios, which rely on predictions.

- **B3**: similar to **B2**, but applying the Stanford enhancer post-hoc over the predicted trees. Scores for Finnish and Latvian were not provided.

4 Results

Table 3 displays our results on the per-language (not per-treebank) test partitions of the shared task data. As explained in §2, for languages with multiple training treebanks available (Czech, Estonian, Dutch, Polish), we preprocessed the raw text of each treebank using the pipeline trained on the largest treebank available for that language (e.g. alpino for Dutch). Also, aforementioned in §3.4, we then trained our parsers on the concatenation of each language’s treebanks, so that we could parse at the language level (as opposed to treebank). Though we observed scant differences between the two preprocessing pipelines, it was UDPIPE that produced fewer validation errors. As such, we adopted it as the main preprocessor for our official submission.

It is apparent in Table 3 that the unconnected graph issue (described in §3.3) severely affected our official submission to the shared task (observed in the **official** column). After diagnosing and fixing this problem, we observed an improvement of 14.1 ELAS, which is consistent with our scores on the treebanks’ development sets. With this in mind, our (fixed) parser tends to perform in a generally stable fashion across languages, with an average ELAS of 76.48 and standard deviation of 6.86. Among our highest scoring languages are Bulgarian, French, and Italian—the former two of which are corroborated by the strong **B1** baseline. Indeed, Tamil is the notable exception among all results, with 56.85 ELAS. We surmise that treebank size is the biggest factor in this degradation of performance, as Tamil has, by far, the smallest treebank at 400 sentences. As such, our parser has comparatively fewer graph samples to train on than it would for some other languages.

When comparing against the organizer-provided baselines, we see a strong improvement in using our system over both **B2** and **B3** systems. This is encouraging, as it demonstrates the benefit of parsing enhanced dependency graphs directly, rather
than relying on predicted trees to accurately relay the enhanced structure \((B1)\) or employing a heuristic-driven post-processor to derive it \((B2)\). Furthermore, though the organizers consider \(B1\) as an indirect upper bound due to the gold-standard tokenization and dependency structure employed therein, we can nonetheless observe an advantage in using our parser for some languages. These are Arabic (+2.16 ELAS), Finnish (+3.32), Italian (+4.46), and Ukranian (+0.86). Again, this is promising, given that our parser does not rely on any tree structure in order to parse graphs.

### 4.1 Pre-processing Analysis

Since the test data was provided in a raw, untokenized format, we were interested in the extent of accuracy loss we might observe when relying on off-the-shelf pre-processors. Table 4 displays these results over the development data. It is clear that when we employ predicted segmentation, etc. using either STANZA or UDPIPE pipelines, we observe a slight degradation in accuracy, as compared to the gold data. Omitting Czech, Estonian, Dutch, and Polish (which had several associated treebanks), all other languages degrade by an average of 2.00 ELAS for STANZA and 2.32 for UDPIPE. Though one typically expects such a degradation when evaluating with predicted segmentation, we did observe some unreasonably large gaps in accuracy: namely for Arabic \((-4.02, -8.32\) for STANZA and UDPIPE, respectively) and Tamil \((-12.19, -8.59\). The latter can likely be explained via its small training set, which undoubtedly affects all components of the preprocessing pipeline.

When we examine the scores for all multitreebank languages, we do not notice a large difference between gold and predicted tokenization—whicj we expect to be different across treebanks. Here, we simply choose the one trained on the largest treebank (PicTree for Czech, EDT for Estonian, Alpino for Dutch, and LFG for Polish), as we consider this a simple yet reliable heuristic. However, when generating predictions for the smaller treebanks using the bigger treebank’s preprocessing model, we only notice a notable drop in accuracy for Dutch \((-2.15, -2.54\) for STANZA and UDPIPE, respectively). This indicates that there are likely major differences in the treebanks’ domains or how they are respectively segmented or annotated. In general, however, the differences between gold and predicted tokenization is surprisingly not as large as we expected.

### 5 Conclusion

In this paper, we have described the IWPT 2020 Shared Task submission by the Copenhagen-Uppsala team, consisting of graphs predicted by a transition-based neural dependency graph parser with pre-trained multilingual contextualized embeddings. While not ranked among the top submission according to the official scores, the parser architecture proved effective for the type of dependency graphs exhibited by ED, and after fixing a critical bug we found the scores to improve dramatically and agree with the scores we had observed during development.

We expect that with more resources for BERT fine-tuning, hyperparameter tuning, language-specific pre-trained representations and careful pre- and post-processing, our parser will be a competitive system in this task. However, our contribution is a transition system that can directly handle ED, in a unified transition-based parsing system.

<table>
<thead>
<tr>
<th>Language</th>
<th>STANZA</th>
<th>UDPIPE</th>
<th>Gold Tok.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARABIC</td>
<td>73.66</td>
<td>69.36</td>
<td>77.68</td>
</tr>
<tr>
<td>BULGARIAN</td>
<td>83.46</td>
<td>83.17</td>
<td>83.89</td>
</tr>
<tr>
<td>CZECH</td>
<td>75.60</td>
<td>75.47</td>
<td>76.00</td>
</tr>
<tr>
<td>DUTCH</td>
<td>78.66</td>
<td>78.27</td>
<td>80.81</td>
</tr>
<tr>
<td>ENGLISH</td>
<td>80.79</td>
<td>79.80</td>
<td>82.77</td>
</tr>
<tr>
<td>ESTONIAN</td>
<td>75.43</td>
<td>75.32</td>
<td>75.81</td>
</tr>
<tr>
<td>FINNISH</td>
<td>80.87</td>
<td>80.59</td>
<td>81.89</td>
</tr>
<tr>
<td>FRENCH</td>
<td>86.05</td>
<td>85.29</td>
<td>88.97</td>
</tr>
<tr>
<td>ITALIAN</td>
<td>85.24</td>
<td>85.04</td>
<td>85.52</td>
</tr>
<tr>
<td>LATVIAN</td>
<td>79.00</td>
<td>78.39</td>
<td>79.28</td>
</tr>
<tr>
<td>LITHUANIAN</td>
<td>74.92</td>
<td>74.84</td>
<td>75.51</td>
</tr>
<tr>
<td>POLISH</td>
<td>71.94</td>
<td>73.22</td>
<td>73.63</td>
</tr>
<tr>
<td>RUSSIAN</td>
<td>78.53</td>
<td>78.60</td>
<td>78.87</td>
</tr>
<tr>
<td>SLOVAK</td>
<td>77.54</td>
<td>77.33</td>
<td>79.17</td>
</tr>
<tr>
<td>SWEDISH</td>
<td>78.26</td>
<td>78.18</td>
<td>78.37</td>
</tr>
<tr>
<td>TAMIL</td>
<td>50.66</td>
<td>54.26</td>
<td>62.85</td>
</tr>
<tr>
<td>UKRAINIAN</td>
<td>79.70</td>
<td>79.67</td>
<td>79.89</td>
</tr>
</tbody>
</table>

**Table 4**: Development ELAS for our fixed parser. While in all cases we train the parser on the concatenation of all of a language’s gold treebanks (applicable only to Czech, Dutch, Estonian, and Polish), STANZA and UDPIPE refer to generating predictions on the development data preprocessed by the corresponding pipeline. **Gold Tok.** refers to generating predictions over gold development data (tokenization, etc.).
Acknowledgments

We thank the anonymous reviewers for their helpful comments. ML is funded by a Google Focused Research Award. We acknowledge the computational resources provided by CSC in Helsinki and Sigma2 in Oslo through NeIC-NLPL (www.nlpl.eu).

References


