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,[1] 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).

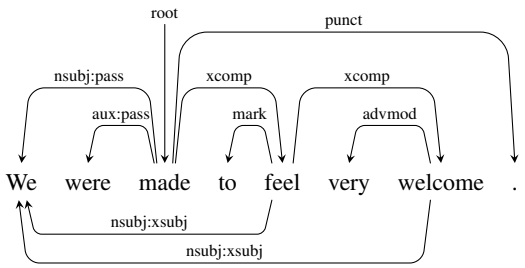

Figure 1: ED for `reviews-077034-0002` from `UD_English-EWT`, containing a control verb (made). Arcs above the sentence are also in BD.

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
[1]Some BD arcs are deleted in ED, e.g., `orphan` arcs.

Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task, pages 236–244
Virtual Meeting, July 9, 2020.

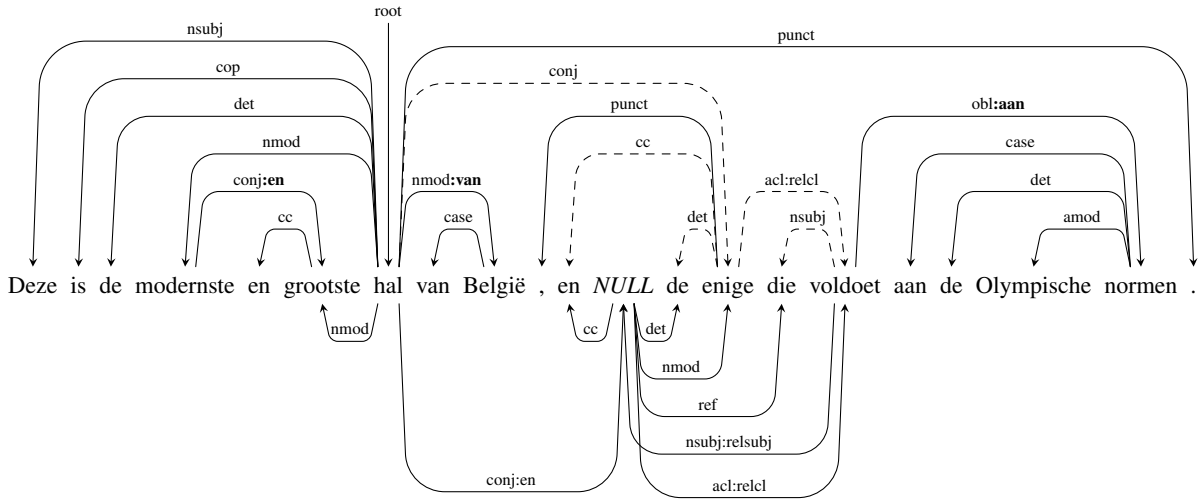

Figure 2: `wiki-3745.p.38.s.5` from `UD_Dutch-LassySmall`, containing a null node *NULL*, not in the original sentence, coordination and case suffixes (**`:en`**, **`:van`**, **`:aan`**), and propagation of conjuncts (hal → grootste). The dashed edges are deleted in ED, and the edges below the sentence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

(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.

	Language			
	CZECH	DUTCH	ESTONIAN	POLISH
combined	78.88	**76.50**	77.01	**82.96**
largest	**83.97**	74.97	**77.61**	82.59

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.

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-

[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

sition 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: {SHIFT, LEFT-EDGE$_l$, RIGHT-EDGE$_l$, REDUCE-0, REDUCE-1, NODE, SWAP and FINISH}. The SHIFT transition pops the first element of the buffer and pushes it onto the stack. The LEFT-EDGE$_l$ and RIGHT-EDGE$_l$ transitions add an arc[6] between the two top items of the stack with label l. We need two different REDUCE transitions to pop the topmost and second topmost items of the stack, which we name REDUCE-0 and a REDUCE-1 respectively. This makes it possible to construct length-2 cycles, which ED allows (and most MRP frameworks do not). The NODE transition inserts a null node as the first element of the buffer, needed to support null nodes. 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, FINISH terminates the transition sequence. A formal definition of the transition set is shown in Figure 3.

Separate EDGE transitions exist for each edge label. Labels containing coordination or case suffixes (such as `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.

NODE transitions, on the other hand, do not select any label or features, since null nodes are only evaluated with respect to their incoming and outgoing edges. All other information is ignored, and thus not predicted by the parser: predicted null nodes are thus only placeholders.

Language	Total	EDGE	w/ suffix
ARABIC	402	395	345
BULGARIAN	197	191	137
CZECH	768	761	702
DUTCH	393	386	336
ENGLISH	300	293	232
ESTONIAN	445	438	381
FINNISH	266	259	210
FRENCH	112	106	59
ITALIAN	281	274	216
LATVIAN	238	232	161
LITHUANIAN	323	317	263
POLISH	676	669	615
RUSSIAN	944	938	861
SLOVAK	266	259	204
SWEDISH	209	202	153
TAMIL	146	140	103
UKRAINIAN	290	283	225

Table 2: Number of transitions for each language.

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

Since LEFT-EDGE$_l$ and RIGHT-EDGE$_l$ 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 REDUCE-0 and REDUCE-1 transitions for nodes that do not have a head yet. We also disallow reducing the root.

For the SWAP transition, we maintain the *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 NODE transitions such that there are no more null nodes than words (although a lower limit would suffice), and EDGE transitions to limit the

[6] For consistency, we keep the transition nomenclature using "EDGE", although they create directed dependency *arcs*. Note that in analogous transitions in some transition systems, such as ARCEAGER (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 EDGE action and use our two REDUCE transitions to pop elements of the stack when necessary.

Before Transition				Transition	After Transition					Condition
Stack	Buffer	Nodes	Arcs		Stack	Buffer	Nodes	Arcs	Terminal?	
Σ	$b \mid B$	V	A	SHIFT	$\Sigma \mid b$	B	V	A	$-$	
$\Sigma \mid s_0$	B	V	A	REDUCE-0	Σ	B	V	A	$-$	$s_0 \neq \text{root} \wedge (\cdot, s_0). \in A$
$\Sigma \mid s_1, s_0$	B	V	A	REDUCE-1	$\Sigma \mid s_0$	B	V	A	$-$	$s_1 \neq \text{root} \wedge (\cdot, s_1). \in A$
Σ	B	V	A	NODE	Σ	$b \mid B$	$V \cup \{b\}$	A	$-$	
$\Sigma \mid s_1, s_0$	B	V	A	LEFT-EDGE$_l$	$\Sigma \mid s_1, s_0$	B	V	$A \cup \{(s_0, s_1)_l\}$	$-$	$s_1 \neq \text{root} \wedge (s_0, s_1)_l \notin A$
$\Sigma \mid s_1, s_0$	B	V	A	RIGHT-EDGE$_l$	$\Sigma \mid s_1, s_0$	B	V	$A \cup \{(s_1, s_0)_l\}$	$-$	$(s_1, s_0)_l \notin A$
$\Sigma \mid s_1, s_0$	B	V	A	SWAP	$\Sigma \mid s_0$	$s_1 \mid B$	V	A	$-$	$s_1 \neq \text{root} \wedge \text{i}(s_1) < \text{i}(s_0)$
[root]	[ ]	V	A	FINISH	[ ]	[ ]	V	A	$+$	

Figure 3: Our transition set. We write the stack with its top to the right and the buffer with its head to the left. $(h, d)_l$ denotes an l-labeled dependency with head h and dependent d. $\text{i}(x)$ is the generated order (see §3.1).

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.

	Transition	Stack	Buffer	Arc added
		[ROOT]	[Deze is (...)]	
1-6	SHIFT	[(...) en grootste]	[hal van (...)]	
7	LEFT-EDGE$_{cc}$	[(...) en grootste]	[hal van (...)]	(grootste, en)$_{cc}$
8	REDUCE-1	[(...) modernste grootste]	[hal van (...)]	
9	RIGHT-EDGE$_{conj:en}$	[(...) modernste grootste]	[hal van (...)]	(grootste, modernste)$_{conj:en}$
10	SHIFT	[(...) grootste hal]	[van België (...)]	
11	LEFT-EDGE$_{nmod}$	[(...) grootste hal]	[van België (...)]	(hal, grootste)$_{nmod}$
12	NODE	[(...) grootste hal]	[*NULL* van (...)]	
13-21		Series of LEFT-EDGE and REDUCE-1 transitions		
22	RIGHT-EDGE$_{root}$	[ROOT hal]	[*NULL* van (...)]	(ROOT, hal)$_{root}$
23	SHIFT	[ROOT hal *NULL*]	[van België (...)]	
24	RIGHT-EDGE$_{conj:en}$	[ROOT hal *NULL*]	[van België (...)]	(hal, *NULL*)$_{conj:en}$
25-26	SHIFT	[(...) van België]	[, en (...)]	
27	LEFT-EDGE$_{case}$	[(...) van België]	[, en (...)]	(België, van)$_{case}$
28	REDUCE-1	[(...) *NULL* België]	[, en (...)]	
29	SWAP	[ROOT hal België]	[*NULL* , (...)]	
30	RIGHT-EDGE$_{nmod:van}$	[(...) hal België]	[*NULL* , (...)]	(hal, België)$_{nmod:van}$
31	REDUCE-0	[ROOT hal]	[*NULL* , (...)]	
32-34	SHIFT	[(...) , en]	[de enige (...)]	
35	SWAP	[(...) *NULL* en]	[, de (...)]	
36	RIGHT-EDGE$_{cc}$	[(...) *NULL* en]	[, de (...)]	(*NULL*, en)$_{cc}$
37	REDUCE-0	[ROOT hal *NULL*]	[, de (...)]	
38-39	SHIFT	[(...) , de]	[enige die (...)]	
40	SWAP	[(...) *NULL* de]	[, enige (...)]	
41	RIGHT-EDGE$_{det}$	[(...) *NULL* de]	[, enige (...)]	(*NULL*, de)$_{det}$
42	REDUCE-0	[ROOT hal *NULL*]	[, enige (...)]	
43-44	SHIFT	[(...) , enige]	[die voldoet (...)]	
45	LEFT-EDGE$_{punct}$	[(...) , enige]	[die voldoet (...)]	(enige, ,)$_{punct}$
46	REDUCE-1	[(...) *NULL* enige]	[die voldoet (...)]	
47	RIGHT-EDGE$_{nmod}$	[(...) *NULL* enige]	[die voldoet (...)]	(*NULL*, enige)$_{nmod}$
48	REDUCE-0	[ROOT hal *NULL*]	[die voldoet (...)]	
49	SHIFT	[(...) *NULL* die]	[voldoet aan (...)]	
50	RIGHT-EDGE$_{ref}$	[(...) *NULL* die]	[voldoet aan (...)]	(*NULL*, die)$_{ref}$
51	REDUCE-0	[ROOT hal *NULL*]	[voldoet aan (...)]	
52	SHIFT	[(...) *NULL* voldoet]	[aan de (...)]	
53	RIGHT-EDGE$_{acl:relcl}$	[(...) *NULL* voldoet]	[aan de (...)]	(*NULL*, voldoet)$_{acl:relcl}$
54	LEFT-EDGE$_{nsubj:relsubj}$	[(...) *NULL* voldoet]	[aan de (...)]	(voldoet, *NULL*)$_{nsubj:relsubj}$
55-69		(...)		
70	RIGHT-EDGE$_{punct}$	[ROOT hal .]	[ ]	(hal, .)$_{punct}$
71-72	REDUCE-0	[ROOT]	[ ]	
73	FINISH	[ROOT]	[ ]	

Figure 4: Oracle transition sequence for the sentence from Figure 2. Consecutive SHIFTs grouped for brevity.

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,[10] 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,

[10] https://github.com/coastalcph/koepsala-parser/commit/1b872ad9fc2652649c11eb0a8622c744c92e8cbb

[11] https://github.com/coastalcph/koepsala-parser/blob/master/config/transition_eud.jsonnet

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

Language	Baselines			Ours	
	B1	*B2*	*B3*	*official*	*fixed*
ARABIC	67.35	46.41	45.16	60.84	69.51
BULGARIAN	85.82	73.74	79.9	68.88	84.49
CZECH	78.44	65.31	63.62	61.11	74.79
DUTCH	82.48	62.97	72.65	62.93	76.92
ENGLISH	84.30	66.83	76.16	65.37	81.05
ESTONIAN	76.38	57.53	54.34	59.07	72.38
FINNISH	78.26	61.71	-	67.54	81.58
FRENCH	97.49	71.14	63.31	67.93	82.76
ITALIAN	80.20	70.33	83.03	69.08	84.66
LATVIAN	79.31	59.14	-	64.75	79.12
LITHUANIAN	74.22	46.78	44.84	56.28	69.09
POLISH	81.59	66.38	65.37	61.34	73.89
RUSSIAN	79.63	68.33	67.80	64.23	78.90
SLOVAK	77.60	60.02	58.05	64.08	77.44
SWEDISH	80.98	62.18	71.53	64.50	78.61
TAMIL	76.29	40.71	40.25	47.44	56.85
UKRAINIAN	77.24	58.73	56.92	64.17	78.10
AVERAGE	79.86	61.07	62.90	62.91	76.48

Table 3: Main results for Enhanced Universal Dependencies shared task (ELAS), as evaluated on the provided test sets. *B1, B2, B3* refer to organizer-provided baseline systems. *official* refers to our official submission, prior to fixing the unconnected graph issue (*fixed*).

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 multi-treebank languages, we do not notice a large difference between gold and predicted tokenization—which we expect to be different across treebanks. Here, we simply choose the one trained on the largest treebank (`FicTree` 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.

Language	STANZA	UDPIPE	**Gold Tok.**
ARABIC	73.66	69.36	77.68
BULGARIAN	83.46	83.17	83.89
CZECH	75.60	75.47	76.00
DUTCH	78.66	78.27	80.81
ENGLISH	80.79	79.80	82.77
ESTONIAN	75.43	75.32	75.81
FINNISH	80.87	80.59	81.89
FRENCH	86.05	85.29	88.97
ITALIAN	85.24	85.04	85.52
LATVIAN	79.00	78.39	79.28
LITHUANIAN	74.92	74.84	75.51
POLISH	71.94	73.22	73.63
RUSSIAN	78.53	78.60	78.87
SLOVAK	77.54	77.33	79.17
SWEDISH	78.26	78.18	78.37
TAMIL	50.66	54.26	62.85
UKRAINIAN	79.70	79.67	79.89
AVERAGE	77.08	76.87	78.88

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).

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 critial 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.

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

Omri Abend and Ari Rappoport. 2013. Universal conceptual cognitive annotation (UCCA). In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 228–238, Sofia, Bulgaria. Association for Computational Linguistics.

Gosse Bouma, Djamé Seddah, and Daniel Zeman. 2020. Overview of the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, Seattle, US. Association for Computational Linguistics.

Jan Buys and Phil Blunsom. 2017. Robust incremental neural semantic graph parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1215–1226, Vancouver, Canada. Association for Computational Linguistics.

Wanxiang Che, Longxu Dou, Yang Xu, Yuxuan Wang, Yijia Liu, and Ting Liu. 2019. HIT-SCIR at MRP 2019: A unified pipeline for meaning representation parsing via efficient training and effective encoding. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*, pages 76–85, Hong Kong. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. Allennlp: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 1–6.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A transition-based directed acyclic graph parser for UCCA. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1127–1138, Vancouver, Canada. Association for Computational Linguistics.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2018. Universal dependency parsing with a general transition-based DAG parser. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 103–112, Brussels, Belgium. Association for Computational Linguistics.

Yijia Liu, Wanxiang Che, Bo Zheng, Bing Qin, and Ting Liu. 2018. An AMR aligner tuned by transition-based parser. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2422–2430, Brussels, Belgium. Association for Computational Linguistics.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*, pages 149–160, Nancy, France.

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 *Proc. of LREC*.

Joakim Nivre, Paola Marongiu, Filip Ginter, Jenna Kanerva, Simonetta Montemagni, Sebastian Schuster, and Maria Simi. 2018. Enhancing universal dependency treebanks: A case study. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 102–107, Brussels, Belgium. Association for Computational Linguistics.

Stephan Oepen, Omri Abend, Jan Hajic, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, Jayeol Chun, Milan Straka, and Zdenka Uresova. 2019. MRP 2019: Cross-framework meaning representation parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*, pages 1–27, Hong Kong. Association for Computational Linguistics.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A Python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Sebastian Schuster and Christopher D. Manning. 2016. Enhanced English Universal Dependencies: An improved representation for natural language understanding tasks. In *Proc. of LREC*. ELRA.

Milan Straka, Jan Hajic, and Jana Straková. 2016. Udpipe: trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 4290–4297.

Milan Straka and Jana Straková. 2017. Tokenizing, pos tagging, lemmatizing and parsing ud 2.0 with udpipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada. Association for Computational Linguistics.

Mingxuan Wang, Jun Xie, Zhixing Tan, Jinsong Su, Deyi Xiong, and Chao Bian. 2018. Neural machine translation with decoding history enhanced attention. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1464–1473, Santa Fe, New Mexico, USA. Association for Computational Linguistics.