ÚFAL MRPipe at MRP 2019: UDPipe Goes Semantic in the Meaning Representation Parsing Shared Task

Milan Straka and **Jana Straková**
Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics
{straka,strakova}@ufal.mff.cuni.cz

Abstract

We present a system description of our contribution to the CoNLL 2019 shared task, Cross-Framework Meaning Representation Parsing (MRP 2019). The proposed architecture is our first attempt towards a semantic parsing extension of the UDPipe 2.0, a lemmatization, POS tagging and dependency parsing pipeline.

For the MRP 2019, which features five formally and linguistically different approaches to meaning representation (DM, PSD, EDS, UCCA and AMR), we propose a uniform, language and framework agnostic graph-to-graph neural network architecture. Without any knowledge about the graph structure, and specifically without any linguistically or framework motivated features, our system implicitly models the meaning representation graphs.

After fixing a human error (we used earlier incorrect version of provided test set analyses), our submission would score third in the competition evaluation. The source code of our system is available at `https://github.com/ufal/mrpipe-conll2019`.

1 Introduction

The goal of the CoNLL 2019 shared task, Cross-Framework Meaning Representation Parsing (MRP 2019; Oepen et al., 2019) is to parse a raw, unprocessed sentence into its corresponding graph-structured meaning representation.

The MRP 2019 features five formally and linguistically different approaches to meaning representation with varying degree of linguistic and structural complexity:

- **DM:** DELPH-IN MRS Bi-Lexical Dependencies (Ivanova et al., 2012),
- **PSD:** Prague Semantic Dependencies (Hajič et al., 2012; Miyao et al., 2014),
- **EDS:** Elementary Dependency Structures (Oepen and Lønning, 2006),
- **UCCA:** Universal Conceptual Cognitive Annotation (Abend and Rappoport, 2013),
- **AMR:** Abstract Meaning Representation (Banarescu et al., 2013).

In line with the shared task objective to advance uniform meaning representation parsing across distinct semantic graph frameworks, we propose a uniform, language and structure agnostic graph-to-graph neural network architecture which models semantic representation from input sequences. The system is an extension of the UDPipe 2.0, a tagging, lemmatization and syntactic tool (Straka, 2018; Straka et al., 2019).

Our contributions are the following:

- We propose a uniform semantic graph parsing architecture, which accommodates simple directed cyclic graphs, independently on the underlying semantic formalism.
- Our method does not use linguistic information such as structural constraints, dictionaries, predicate banks or lexical databases.
- We added a new extension to UDPipe 2.0, a lemmatization, POS tagging and dependency parsing tool. The semantic extension parses semantic graphs from the raw token input, making use of the POS and lemmas (but not syntax) from the existing UDPipe 2.0.
- As an improvement over UDPipe 2.0, we use the "frozen" contextualized embeddings on the input (BERT; Devlin et al., 2019) in the same way as Straka et al. (2019).

After fixing a human error (we used earlier incorrect version of provided test set analyses), our submission would score third in the competition evaluation.

2 Related Work

Numerous parsers have been proposed for parsing semantic formalisms, including the systems participating in recent semantic parsing shared tasks SemEval 2016 and SemEval 2017 (May, 2016; May and Priyadarshi, 2017) featuring AMR; and SemEval 2019 (Hershcovich et al., 2019) featuring UCCA. However, proposals of general, formalism independent semantic parsers are scarce in the literature.

Hershcovich et al. (2018) propose a general transition-based parser for directed, acyclic graphs, able to parse multiple conceptually and formally different schemes. TUPA is a transition-based top-down shift-reduce parser, while ours, although also based on transitions/operations, models the graph as a sequence of layered, iterative graph-like operations, rather (but not necessarily) in a bottom-up fashion. Consequently, our architecture allows parsing cyclic graphs and is not restricted to single-rooted graphs. Also, we do not enforce any task-specific constraints, such as restriction on number of parents in UCCA or number of children given by PropBank in AMR and we completely rely on the neural network to implicitly infer such framework-specific features.

3 Methods

3.1 Uniform Graph Model

The five shared task semantic formalisms differ notably in specific formal and linguistic assumptions, but from a higher-level view, they universally represent the full-sentence semantic analyses with directed, possibly cyclic graphs. Universally, the semantic units are represented with graph nodes and the semantic relationships with graph edges.

To accommodate these semantic structures, we model them as directed simple graphs $G = (V, E)$, where V is a set of nodes and $E \subseteq \{(x, y) \mid (x, y) \in V^2, x \neq y\}$ is a set of directed edges.[1]

One of the most fundamental differences between the five featured MRP 2019 frameworks lies apparently in the relationship between the graph structure (graph nodes) and the input surface word forms (tokens). In the MRP 2019, this relationship is called *anchoring* and its degree varies from a tight connection between graph nodes being directly corresponding to surface tokens in *Flavor 0* frameworks (DM and PSD) through more relaxed relationship *Flavor 1* (EDS and UCCA) in which arbitrary parts of the sentence can be represented in the semantic graph, to a completely *unanchored* semantic graph of *Flavor 2* in the AMR framework.

To alleviate the need for a framework-specific handling of the anchoring, we broaden our understanding of the semantic graph: We consider the tokens as nodes and the anchors (connections from the graph nodes to tokens) as regular edges, thus the anchors are naturally learned jointly with the graph without an explicit knowledge of the underlying semantic formalism.

In order to represent anchors as regular edges in the graph, the input tokenization needs to be consistent with the annotated anchors: each anchor must match one or multiple input tokens. In order to achieve the exact anchor-token(s) match, we created a simple tokenizer. The tokenizer is uniform for all frameworks with a slight change to capture UCCA's fine-grained anchoring; see Figure 1 for the pseudocode.[2]

Furthermore, to represent anchors as edges, the anchors have to be annotated in the data, which is not the case for AMR. We therefore utilize externally generated anchoring from the JAMR tool (Flanigan et al., 2016).[3]

3.2 Graph-to-graph Parser

We propose a general graph-to-graph parser which models the graph meaning representation as a sequence of layered group transformations from input from input sequence to meaning graphs. A schematic overview of our architecture is presented in Figure 2.

Having reduced the task to a graph-to-graph transformation modeling, we iteratively build the graph from its initial state (a set of isolated nodes – tokens) by alternating between two layer-wise transformations:

1. **AddNodes**: The first operation creates new nodes and connects them to already existing

[1]Specifically, our graphs are directed and allow cycles. Furthermore, they are *simple* graphs, not *multigraphs*.

[2]Instead of generating tokens consistent with the anchors, the anchoring edges could be allowed to refer only to a part of a token (for example by having two attributes *first anchored token character* and *last anchored token character*), which is an approach we plan to adopt in the future.

[3]We plan to model the anchors jointly using an attention mechanism (Zhang et al., 2019).

```
 1.  Any single non-space character
2a.  UCCA: \w+[$]?
2b.  other: \w(\w-[^-\s]|&|/|'S\w|'[A-RT-Z]|[.](?=.*\w)\w|\d)*[$]?;
     \d+-\d+; \d+,\d+; \d+,\d+,\d+
 3.  --+; `+; '+; [.]+; !+
 4.  n't;'s;'d;'m;'re;'ve;'ll
 5.  Split the following word into two tokens: would|n't; could|n't; ca|n't; is|n't; are|n't; ai|n't;
     was|n't; were|n't; do|n't; does|n't; did|n't; should|n't; have|n't; has|n't; had|n't;
     wo|n't; might|n't; need|n't; can|not; wan|na; got|ta
```

Figure 1: Tokenizer pseudocode as a sequence of regular expressions. Expressions with higher number override previous ones.

nodes. Specifically, for each already existing node we decide whether to a) create a new node and connect it as a parent, b) create a new node and connect it as a child, c) do nothing. When a new node is created, its label and all its properties are generated too. Intuitively, anchors are modeled in the first step from the initial set of individual nodes (tokens) and in the next steps, higher-layer nodes are modeled. As a special case, **AddNodes** is relatively simple for the Flavor 0 frameworks (DM and PSD): zero or one node is created for every token in the first and only **AddNodes** iteration. This is illustrated in Table 1, which shows node coverage after performing a fixed number of **AddNodes** iterations, reaching 100% after one **AddNodes** iteration in DM and PSD.

2. **AddEdges**: The second operation creates edges between the new nodes and any other existing nodes (both old and new) using a classifier for each pair of nodes. Any number of edges can be connected to a newly created node.

At the end of each iteration, the created nodes and edges are frozen and the computation moves to its next iteration. We describe the crucial part of the graph modeling, **token**, **node** and **edge representation**, in Section 3.4.

An example of a graph step by step build-up is shown in Figure 2.

In contrast to purely sequential series of single transitions, such as adding a new edge in one step, adding new nodes and edges in a layer-wise fashion improves runtime performance and might avoid error accumulation by performing many independent decisions. On the other hand, we assume that creating nodes from a single existing one might be problematic, especially if the graph has constituency structure.

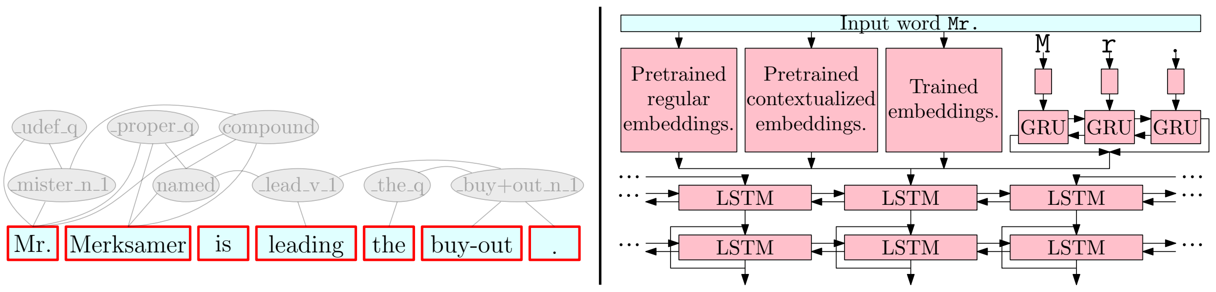
3.2.1 Creating AddNodes Operations

For training, a sequence of the **AddNodes** operations must be created. For this purpose, we define an ordering of the graph nodes which guides the graph traversal. The initial order of the isolated graph nodes set (tokens) is left to right, the first token being the first to be visited. The other graph nodes' ordering is then induced by the order of creation.

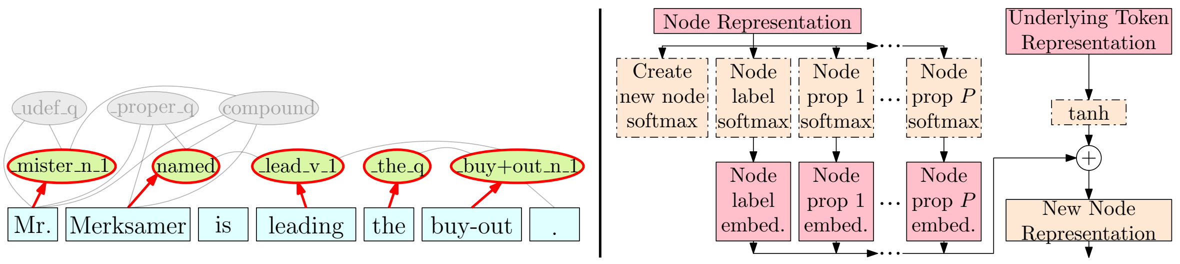
Given a training graph, we then generate a sequence of **AddNodes** operations. In every iteration, we traverse all existing nodes in the graph in the above defined order and for each node, we consider all its not-yet-created neighbors, from which we choose the one which is "in the lowest layer". This is motivated by our intention to build the graph in a bottom-up fashion. Specifically, we choose such a node which has the smallest number of token descendants (based on the assumption that nodes in the lower levels tend to govern less descendants than the nodes in the higher levels), and if there are several such nodes, the one where the token descendant indices are smallest in the ordering. Finally, we favour creating parents to creating children, and if a node can be created as a parent, we never create it as a child.

As a special case, the first iteration always traverses the set of isolated nodes (tokens) and connects their immediate parents with the anchor-defined edges. For DM and PSD frameworks, this is the first and only iteration of the **AddNodes** operations.

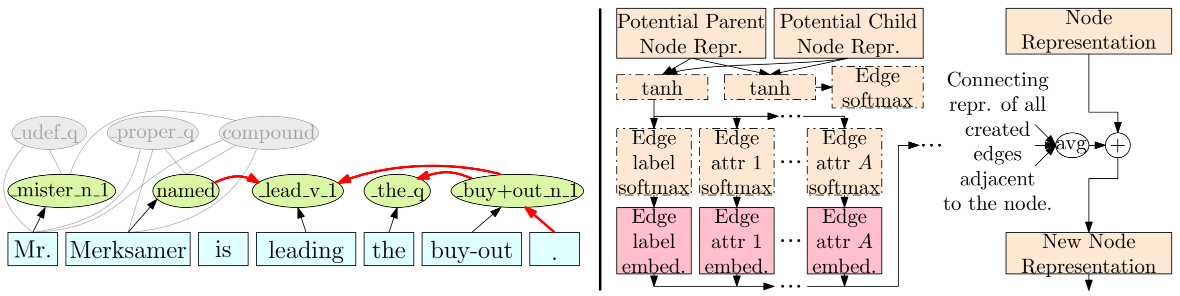
The number of required iterations to generate all nodes and construct complete graphs is presented in Table 1. Performing three iterations is enough to cover more than 99% of nodes in all frameworks, but EDS and AMR frameworks sometimes require more than 10 iterations to generate a full graph.



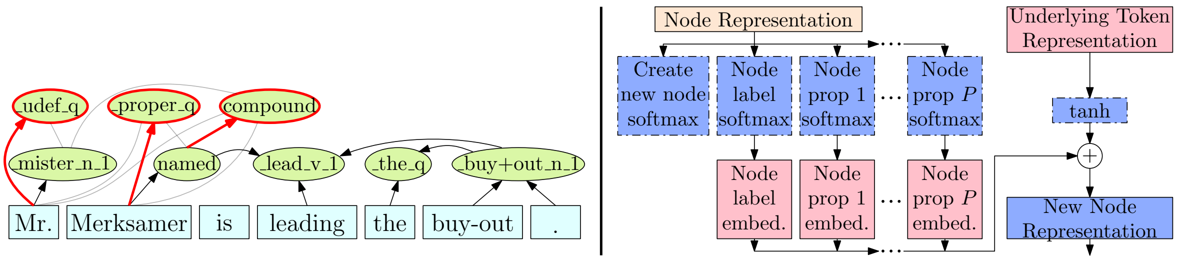
(a) Left: Initial configuration with tokens only. Right: Token representation encoder architecture.



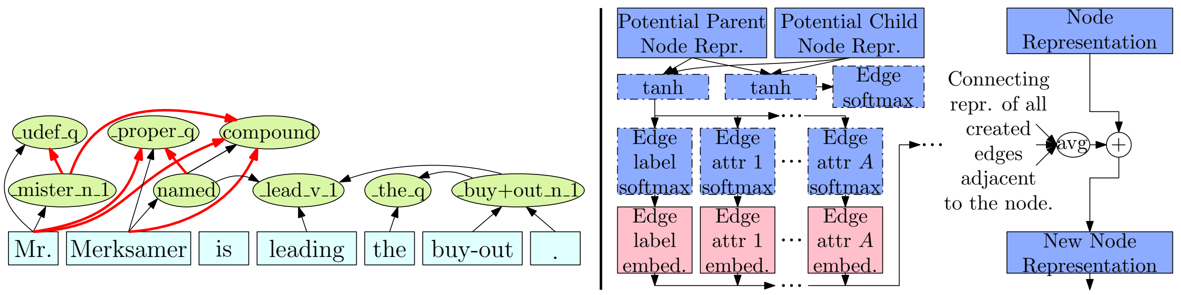
(b) Left: First **AddNodes** operation. Right: Architecture of the new node classifier and representation encoder.



(c) Left: First **AddEdges** operation. Right: Architecture of the edge classifier and updated node representation encoder.



(d) Left: Second **AddNodes** operation. Right: Architecture of the new node classifier and representation encoder.



(e) Left: Second **AddEdges** operation. Right: Architecture of the edge classifier and updated node representation encoder.

Figure 2: Our graph-to-graph architecture schematic overview and an example of semantic graph build-up for the sentence *"Mr. Merksamer is leading the buy-out."* from the EDS framework (Oepen and Lønning, 2006). Note that the weights for all classification layers and for all displayed fully connected layers (displayed with dashed border) are different for every iteration of **AddNodes/AddEdges** operations.

Framework		Iterations									
		1	2	3	4	5	6	7	8	9	10
DM	Nodes	100.00%									
	Graphs	100.00%									
PSD	Nodes	100.00%									
	Graphs	100.00%									
EDS	Nodes	69.18%	97.18%	99.31%	99.64%	99.90%	99.95%	99.97%	99.99%	100.00%	100.00%
	Graphs	2.15%	59.31%	91.68%	93.09%	98.84%	99.46%	99.55%	99.87%	99.99%	99.99%
UCCA	Nodes	69.63%	97.57%	99.87%	99.97%	100.00%	100.00%				
	Graphs	0.00%	43.72%	97.29%	99.19%	99.92%	100.00%				
AMR	Nodes	78.23%	96.15%	99.01%	99.69%	99.88%	99.94%	99.96%	99.97%	99.98%	99.99%
	Graphs	19.73%	74.58%	93.48%	98.18%	99.49%	99.86%	99.94%	99.96%	99.97%	99.98%

Table 1: Coverage of training graphs after a fixed number of the layer-wise iterations. Rows labeled "Nodes" show percentage of covered nodes. Rows labeled "Graphs" show percentage of complete graphs.

During inference, we currently perform a fixed number of iterations of **AddNodes** and **AddEdges** operations; we use one iteration for DM and PSD, two iterations for UCCA and AMR, and three iterations for EDS. Alternatively, we could allow a dynamic number of iterations, stopping when **AddNodes** generates no new nodes.

3.3 Node Labels and Properties Encoding

Besides the graph structure, node labels and properties must also be modeled. For some node labels or properties, it might be beneficial to generate them relatively to a token. For example, when creating a lemma *look* from a token *looked*, it might be easier to generate it as a rule *remove the last two token characters* instead of generating *look* directly. Such approach was taken by UDPipe lemmatizer (Straka et al., 2019), which produced the best results in lemmatization in Task 2 of the SIGMORPHON 2019 Shared Task.

We adopt this approach, and generate all node labels and properties using a simple classification into a collection of rules. Each rule can either generate an independent value (which we call *absolute encoding*) or it describes how a value should be created from a token (which we call *relative encoding*). For detailed description of the relative encoding rules, please refer to Straka et al. (2019). In short, the lemmas in UDPipe are generated by classifying into a set of character edit scripts performed on the prefix and suffix. First, a common root is found between the input and the output (word form and lemma). If there is no common character, the lemma is considered irregular and an *absolute encoding* is used. Otherwise, the shortest edit script is computed for the prefix and suffix.

In our setting, however, we need to extend the UDPipe approach in two directions. First, some properties like *pos* should never be relatively encoded. Therefore, during data loading, we consider both allowing and disallowing relative encoding, and choose the approach yielding the smaller number of classes. As Table 2 indicates, even such a simple heuristic seems satisfactory.

Framework	Property	Absolutely encoded values	Relatively encoded values
DM	label	26 907	**1 086**
	pos	**38**	356
	frame	**468**	2613
PSD	label	32 284	**774**
	pos	**42**	314
	frame	**5 294**	8 868
EDS	label	15 905	**4 339**
	carg	13 667	**427**
UCCA	—	—	—
AMR	label	14 554	**6 278**
	op1	7 377	**1 402**
	op2	3 673	**545**
	op3	1 149	**242**
	op4	482	**113**
	op5	245	**56**
	ARG1	48	**30**
	ARG2	127	**68**
	ARG3	22	**20**
	quant	885	**603**
	value	861	**590**
	time	**110**	111
	year	153	**58**
	li	56	**40**
	mod	79	**33**
	day	**31**	57
	month	**14**	17
	...	...	...

Table 2: Cardinality of absolute and relative encoded node properties in all frameworks. The chosen encoding is displayed in **bold**.

Second, compared to lemmatization, where the lemma and the original form are single words, in our setting both the property and the anchored tokens can be a sequence of words (e.g., "Pierre

Vinken"). We overcome this issue by encoding each word of a property independently, and for every property word, we choose a subsequence of anchoring tokens which yields the shortest relative encoding.

3.4 Graph Representation

Token Encoder. The input representation is a sequence of tokens encoded as a concatenation of word and character-level word vectors:

- trainable word embeddings (WE),
- character-level word embeddings (CLE): bidirectional GRUs in line with Ling et al. (2015). We represent every Unicode character with a vector of dimension 256, and concatenate GRU output for forward and reversed word characters. The character-level word embeddings are trained together with the network.
- pre-trained FastText word embeddings of dimension 300 (Mikolov et al., 2018),[4]
- pre-trained ("frozen") contextual BERT embeddings of dimension 768 (Devlin et al., 2019).[5] We average the last four layers of the BERT model and we produce a word embedding for a token as an average of the corresponding BERT subword embeddings.

  Contextualized embeddings have recently been shown to improve performance of many NLP tasks, see for example Straka et al. (2019) in the context of UDPipe and POS tagging, lemmatization and dependency parsing. Therefore, we expected that utilization of BERT embeddings would improve results considerably, which was the case, as demonstrated in Section 4.1.

Furthermore, the input tokens could be processed by a POS tagger, lemmatizer, dependency parser or a named entity recognizer. If such analyses are available, they can be used as additional embeddings of input tokens. Specifically, we utilize the POS tags and lemmas provided in the shared task. We did not experiment with dependency parses, which we plan to do in the future. Furthermore, we tried utilizing the Illinois Named Entity Tagger (Ratinov and Roth, 2009), but it did not improve our results.

[4]`https://fasttext.cc/docs/en/english-vectors.html`

[5]We use the `Base English Uncased` model from `https://github.com/google-research/bert`.

All available embeddings for a token are concatenated and processed with two bidirectional LSTM layers with residual connections.

Node Encoder. A node is represented by a concatenation of these features:

- the (transitively) attaching token representation (every node has exactly one token which generated it using the **AddNodes** operations), transformed by a dense layer followed by $\tanh$ nonlinearity; every **AddNodes** iteration has its own dense layer weights,
- the node *label* and *properties* embeddings,
- an average of edge representations of all connected edges.

A natural extension would be to represent all node's descendants instead of the one token generating this node through a sequence of **AddNodes**, because the current implementation seems to generate suboptimal representations in later iterations. We leave a proper way of propagating all information through the graph as our future work.

Edge Representation. An edge is represented by a sum of its *label* and *attributes* embeddings.

3.5 Decoders

In the **AddNodes** operation, we employ the following classification decoders, each utilizing the node representation and consisting of a fully connected layer followed by a softmax activation:

- decide among three possibilities, whether to a) add a node as a parent, b) add a node as a child, or c) do nothing;
- generate node label;
- for each property, generate its value (or a special class `NONE`).

During training, we sum the losses of the decoders, apart from the situation when no new node is created, in which case we ignore the label and properties losses.

In the **AddEdges** operation, we consider all edges to and from the newly created nodes. Utilizing all suitable pairs of nodes, we decide for each pair separately whether to add an edge or not.

Although biaffine attention seems to be the preferred architecture for dependency parsing recently (Zeman et al., 2018), in our experiments it performed poorly when we used it for deciding whether to add an edge between any pair of nodes individually. Our hypothesis is that the range of the biaffine attention output is changing rapidly.

That is not an issue when the outputs "compete" with each other in a softmax layer, but is problematic when we compare each with a fixed threshold.

Consequently, we utilized a Bahdanau-like attention (Bahdanau et al., 2014) instead. Specifically, we pass potential parent and child nodes' representations through a pair of fully connected layers with the same output dimensionality, sum the results, apply a $\tanh$ nonlinearity, and attach a binary classifier (a fully connected layer with two outputs and a softmax activation) indicating whether the edge should be added.[6]

In order to predict edge label and attributes, we repeat the same attention process (pass potential parent and child nodes' representation through a different pair of fully connected layers, sum and $\tanh$), and attach classifiers for edge labels and as many edge attributes as present in the data.

Lastly, in order to predict *top* nodes, we employ a sigmoid binary classifier processing the final node representations.

Finally, every iteration of **AddNodes** and **AddEdges** operations has invididual set of weights for all layers described in this section.

3.6 Training

We implemented the described architecture using TensorFlow 2.0 beta (Agrawal et al., 2019). The eager evaluation allowed us to construct inputs to **AddNodes** and **AddEdges** for every batch specifically, so we could easily handle dynamic graphs.

We trained the network using a lazy variant of Adam optimizer (Kingma and Ba, 2014)[7] with $\beta_2 = 0.98$, for 10 epochs with a learning rate of 10^{-3} and for 5 additional epochs with a learning rate 10^{-4} (the difference being UCCA which used 15 and 10 epochs, respectively, because of considerably smaller training data). We utilized a batch size of 64 graphs.[8] The training time on a single GPU was 1-4 hours for DM, PSD, EDS and UCCA, and 10 hours for AMR.

For replicability, we also describe the used hyperparameters in detail. The only differences among the frameworks were:

- slightly different tokenizer for UCCA (Fig 1),
- larger number of training epochs for UCCA,
- number of layer-wise iterations: 1, 1, 3, 2, 2 for DM, PSD, EDS, UCCA and AMR, respectively.

In the encoder, we utilized trainable embeddings of dimension 512, and trainable character-level embeddings using character embeddings of size 256 and a single layer of bidirectional GRUs with 256 units. We processed token embeddings using two layers of bidirectional LSTMs with residual connections and a dimension of 768. The node representations also had dimensionality 768, as did node label and properties embeddings. We employed dropout with rate 0.3 before and after every LSTM layer and on all node representations, and utilized also word dropout (zeroing the whole WE for a given word) with a rate of 0.2. In the **AddEdges** operation, all attention layers have a dimensionality of 1024.

3.7 Data Preprocessing

We created two train/dev splits from the training data provided by the organizers: Firstly, a 90%/10% train/dev split was used to train the model and tune the hyperparameters of the competition entry. For the ablation experiments in the post-competition phase, we later tried a 99%/1% train/dev split, which improved the results only marginally, as shown in Section 4.1.

We further used the provided morphological annotations and the JAMR anchoring for the AMR framework (Flanigan et al., 2016).

4 Results

We present the overall results of our system in Table 3. Please note that our official shared task submission contained an error – test data companion analyses had been updated during the evaluation phase, but we used the original incorrect ones for DM, PSD and EDS frameworks. The error was discovered only after the official deadline, at which point we sent a bugfix submission using the same trained models, the only difference being the utilization of the correct test data analyses during prediction. We present both these submissions in the Table 3, but refer only to the bugfix submission from now on.

The overall results of our system using the official MRP metric are present in Table 3. All reported scores are macro-averaged F1 scores of all

[6] We always add an edge generated in the **AddNodes** operation independently on the prediction for that edge in the **AddEdges** operation.

[7] `tf.contrib.opt.lazyadamoptimizer`

[8] Because we trained on a 8GB GPU, we actually needed to process two batches of size 32 and only then perform parameter update using summed gradients.

System	Tops		Labels		Properties		Anchors		Edges		Attributes		All	
Original ST submission	75.12%	6	63.99%	7	56.53%	6	69.53%	6	62.17%	7	7.85%	4	74.74%	6
Bugfix ST submission	81.47%	6	**73.06%**	**1**	**69.95%**	**1**	77.23%	3	73.89%	5	7.87%	4	83.96%	3
99% training data	80.59%	6	**73.06%**	**1**	**70.18%**	**1**	77.35%	3	74.27%	5	7.96%	4	84.14%	3
No BERT embeddings	70.50%	8	70.71%	4	67.01%	4	76.02%	4	65.02%	6	5.30%	6	78.99%	5
Ensemble	81.13%	6	**73.39%**	**1**	**70.82%**	**1**	77.57%	3	75.85%	4	8.28%	3	85.05%	3
HIT-SCIR	*90.41%*	*2*	*70.85%*	*3*	***69.86%***	***1***	*77.61%*	*2*	***79.37%***	***1***	***12.40%***	***1***	***86.20%***	***1***
SJTU–NICT	***91.50%***	***1***	*71.24%*	*2*	*68.73%*	*2*	***77.62%***	***1***	*77.74%*	*2*	*9.40%*	*2*	*85.27%*	*2*
Soochow	*86.01%*	*5*	*69.50%*	*4*	*68.24%*	*3*	*77.11%*	*3*	*76.85%*	*3*	*8.16%*	*3*	*83.96%*	*3*
Saarland	*86.70%*	*4*	***71.33%***	***1***	*61.11%*	*5*	*75.08%*	*5*	*75.01%*	*4*	—		*81.87%*	*4*

Table 3: Overall results, macro-averaged on all frameworks. We present F1 scores and ranks compared to official ST submissions. Results with rank 1 are typeset in **bold**, best results in each column have gray background.

five frameworks. The results for individual frameworks are presented in Table 4.

Our bugfix submission would score third in in the macro-averaged *all* metric. Overall, our system reaches high accuracy in node *labels* and *properties* prediction, ranking first in both of them. These predictions employ the relative encoding extended from UDPipe and demonstrate its effectiveness.

The weakest points of our system are the *top* nodes prediction and *edges* prediction. We hypothesise that the lower performance of the **AddEdges** operation could be improved by better node representation (i.e., including all dependent tokens of a node, not only the one token generating the node) and by a better edge prediction architecture (i.e., global decision over edge connection in the context of all graph nodes instead of considering only the current node pair).

Framework-wise, our system would achieve ranks 5, 4, 4, 4 and 4 on DM, PSD, EDS, UCCA and AMR, respectively, showing relatively balanced performance. The largest absolute performance gap of our system occurs on UCCA, where we reach 8 percent points lower score than the best system, which is supposedly caused by the fact that there are no *labels* and *properties* which our system excels in predicting, and also by the constituency structure of the UCCA graphs which we represent poorly.

4.1 Ablation Experiments

Given that our submission utilized only 90% of the available training data, we also evaluated a variant employing 99% of the training data, keeping the last 1% for error detection. However, as Tables 3 and 4 show, the results are nearly identical.

In order to asses the BERT embeddings effect, we further evaluated a version of our system without them. The macro-averaged *all* performance without BERT embeddings is substantially lower, 79% compared to 84%. Generally all metrics decrease without BERT embeddings, showing that contextual embeddings help "everywhere".

Lastly, we evaluated performance of an 5-model ensemble. Each model was trained using 99% of the training data and utilized different random initialization. The system performance increased by more than 1 percent point. Although the overall rank of the ensemble is unchanged, the rank on individual frameworks increased from 5 to 2 on DM, from 4 to 1 on PSD, 4 to 3 on EDS and 4 to 2 on AMR. As with the non-ensemble system, the weakest point of our solution are the *edge* predictions, which rank 8, 7, 6, 4 and 3 on DM, PSD, EDS, UCCA and AMR, respectively.

5 Conclusions

We introduced a uniform graph-to-graph architecture for parsing into semantic graphs. The model implicitly learns the linguistic information and the graph structure without the need for any specific hand-crafted or structural knowledge and is suitable for any directed graph, including graphs with cycles. In contrast to a transition-based system, we build the graph in a layer-wise fashion, with operations joined in groups.

Acknowledgments

The work described herein has been supported by OP VVV VI LINDAT/CLARIN project (CZ.02.1.01/0.0/0.0/16_013/0001781) and it has been supported and has been using language resources developed by the LINDAT/CLARIN project (LM2015071) of the Ministry of Education, Youth and Sports of the Czech Republic.

We thank the anonymous reviewers for their insightful comments.

System	Tops	Labels	Properties	Anchors	Edges	Attributes	All
Bugfix ST submission	87.39% 8	**97.29% 1**	94.50% 5	99.02% 6	88.32% 8	—	94.66% 5
99% training data	88.36% 8	**97.38% 1**	94.57% 5	99.04% 6	88.47% 8	—	94.75% 4
No BERT embeddings	80.70% 9	96.24% 2	92.19% 7	98.45% 8	80.06% 10	—	91.75% 8
Ensemble	89.06% 7	**97.51% 1**	94.86% 4	99.12% 3	89.72% 8	—	95.17% 2
HIT-SCIR	*92.65% 3*	*93.00% 4*	*95.33% 3*	***99.28% 1***	*92.54% 2*	—	*95.08% 2*
SJTU–NICT	*93.26% 2*	*94.89% 3*	*95.49% 2*	*99.27% 2*	*92.39% 3*	—	***95.50% 1***
Soochow	*91.13% 6*	*90.27% 8*	*91.51% 7*	*98.16% 8*	*89.84% 7*	—	*92.26% 7*
Saarland	*85.87% 8*	***96.82% 1***	*93.55% 5*	*99.05% 5*	*90.95% 6*	—	*94.69% 4*

(a) DM framework

System	Tops	Labels	Properties	Anchors	Edges	Attributes	All
Bugfix ST submission	94.48% 6	**95.94% 1**	92.61% 2	99.00% 4	76.06% 7	—	90.96% 4
99% training data	86.49% 9	**96.05% 1**	92.70% 2	99.00% 3	76.37% 7	—	90.89% 4
No BERT embeddings	67.57% 12	95.14% 3	90.72% 7	98.47% 8	68.22% 10	—	87.58% 8
Ensemble	87.35% 8	**96.19% 1**	93.04% 2	99.02% 3	78.20% 7	—	**91.51% 1**
HIT-SCIR	*96.03% 3*	*89.30% 5*	***93.10% 1***	***99.12% 1***	*79.65% 3*	—	*90.55% 4*
SJTU–NICT	***96.30% 1***	*93.14% 4*	*91.57% 5*	*99.11% 2*	***80.27% 1***	—	*91.19% 3*
Soochow	*86.55% 8*	*84.51% 8*	*85.03% 8*	*97.51% 8*	*75.22% 7*	—	*85.56% 8*
Saarland	*93.50% 6*	*95.21% 2*	*92.20% 4*	*99.00% 3*	*78.32% 6*	—	***91.28% 1***

(b) PSD framework

System	Tops	Labels	Properties	Anchors	Edges	Attributes	All
Bugfix ST submission	82.82% 6	89.99% 3	**91.21% 1**	92.67% 4	84.76% 7	—	89.12% 4
99% training data	83.79% 6	90.19% 3	**91.19% 1**	92.88% 4	85.09% 6	—	89.37% 4
No BERT embeddings	73.91% 8	84.52% 5	85.76% 3	89.08% 5	76.73% 7	—	83.43% 7
Ensemble	84.59% 6	90.86% 2	**92.00% 1**	93.52% 3	86.55% 6	—	90.29% 3
HIT-SCIR	*85.23% 5*	*89.45% 3*	*89.54% 2*	*94.29% 2*	*88.77% 3*	—	*90.75% 2*
SJTU–NICT	*87.72% 3*	*89.42% 4*	*77.53% 4*	*93.37% 3*	*87.82% 4*	—	*89.90% 3*
Soochow	*89.94% 2*	***91.20% 1***	***89.72% 1***	***94.86% 1***	*89.66% 2*	—	***91.85% 1***
Saarland	*86.31% 4*	*90.61% 2*	*78.99% 3*	*86.55% 6*	***90.96% 1***	—	*89.10% 4*

(c) EDS framework

System	Tops	Labels	Properties	Anchors	Edges	Attributes	All
Bugfix ST submission	62.51% 9	—	—	95.44% 2	59.45% 4	39.35% 4	73.24% 4
99% training data	63.53% 9	—	—	95.80% 2	60.51% 4	39.81% 4	73.95% 4
No BERT embeddings	59.40% 10	—	—	94.11% 5	48.70% 8	26.52% 6	66.90% 7
Ensemble	63.28% 9	—	—	96.19% 2	62.14% 4	41.39% 3	75.22% 4
HIT-SCIR	***100.00% 1***	—	—	*95.36% 3*	***72.66% 1***	***61.98% 1***	***81.67% 1***
SJTU–NICT	*95.31% 5*	—	—	***96.36% 1***	*65.56% 3*	*47.00% 2*	*77.80% 3*
Soochow	*99.56% 3*	—	—	*95.02% 4*	*67.74% 2*	*40.80% 3*	*78.43% 2*
Saarland	*80.95% 8*	—	—	*90.81% 6*	*52.66% 6*	—	*67.55% 6*

(d) UCCA framework

System	Tops	Labels	Properties	Anchors	Edges	Attributes	All
Bugfix ST submission	80.17% 6	82.09% 4	71.44% 5	—	60.83% 6	—	71.83% 4
99% training data	80.77% 6	81.69% 4	72.45% 4	—	60.93% 6	—	71.73% 5
No BERT embeddings	70.91% 8	77.67% 6	66.36% 6	—	51.39% 8	—	65.29% 7
Ensemble	81.39% 6	82.40% 3	74.21% 4	—	62.65% 3	—	73.03% 2
HIT-SCIR	*78.15% 7*	*82.51% 2*	*71.33% 5*	—	*63.21% 2*	—	*72.94% 2*
SJTU–NICT	*84.88% 4*	*78.78% 5*	***79.08% 1***	—	*62.64% 3*	—	*71.97% 3*
Soochow	*62.86% 9*	*81.53% 4*	*74.96% 3*	—	*61.78% 5*	—	*71.72% 5*
Saarland	***86.89% 1***	*74.02% 6*	*40.79% 7*	—	*62.16% 4*	—	*66.72% 6*

(e) AMR framework

Table 4: Results on individual frameworks. We present F1 scores and ranks compared to official ST submissions. Results with rank 1 are typeset in **bold**, best results in each column have gray background.

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.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. 2019. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. *arXiv e-prints*, page arXiv:1903.01855.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. 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.

Jeffrey Flanigan, Chris Dyer, Noah A. Smith, and Jaime Carbonell. 2016. CMU at SemEval-2016 task 8: Graph-based AMR parsing with infinite ramp loss. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1202–1206, San Diego, California. Association for Computational Linguistics.

Jan Hajič, Eva Hajičová, Jarmila Panevová, Petr Sgall, Ondřej Bojar, Silvie Cinková, Eva Fučíková, Marie Mikulová, Petr Pajas, Jan Popelka, Jiří Semecký, Jana Šindlerová, Jan Štěpánek, Josef Toman, Zdeňka Urešová, and Zdeněk Žabokrtský. 2012. Announcing Prague Czech-English dependency treebank 2.0. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*, pages 3153–3160, Istanbul, Turkey. European Languages Resources Association (ELRA).

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2018. Multitask parsing across semantic representations. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 373–385, Melbourne, Australia. Association for Computational Linguistics.

Daniel Hershcovich, Zohar Aizenbud, Leshem Choshen, Elior Sulem, Ari Rappoport, and Omri Abend. 2019. SemEval-2019 task 1: Cross-lingual semantic parsing with UCCA. In *Proceedings of the 13th International Workshop on Semantic Evaluation*, pages 1–10, Minneapolis, Minnesota, USA. Association for Computational Linguistics.

Angelina Ivanova, Stephan Oepen, Lilja Øvrelid, and Dan Flickinger. 2012. Who did what to whom?: A contrastive study of syntacto-semantic dependencies. In *Proceedings of the Sixth Linguistic Annotation Workshop*, LAW VI '12, pages 2–11, Stroudsburg, PA, USA. Association for Computational Linguistics.

Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W. Black, and Isabel Trancoso. 2015. Finding function in form: Compositional character models for open vocabulary word representation. *CoRR*.

Jonathan May. 2016. SemEval-2016 task 8: Meaning representation parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation, SemEval@NAACL-HLT 2016, San Diego, CA, USA, June 16-17, 2016*, pages 1063–1073. The Association for Computer Linguistics.

Jonathan May and Jay Priyadarshi. 2017. SemEval-2017 task 9: Abstract meaning representation parsing and generation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 536–545, Vancouver, Canada. Association for Computational Linguistics.

Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in Pre-Training Distributed Word Representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.

Yusuke Miyao, Stephan Oepen, and Daniel Zeman. 2014. In-House. An ensemble of pre-existing off-the-shelf parsers. In *Proceedings of the 8th International Workshop on Semantic Evaluation*, page 63 – 72, Dublin, Ireland.

Stephan Oepen, Omri Abend, Jan Hajič, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, and Milan Straka. 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 – 20, Hong Kong, China.

Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-based MRS banking. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy. European Language Resources Association (ELRA).

Lev Ratinov and Dan Roth. 2009. Design Challenges and Misconceptions in Named Entity Recognition. In *Proc. of the Conference on Computational Natural Language Learning (CoNLL)*.

Milan Straka. 2018. UDPipe 2.0 Prototype at CoNLL 2018 UD Shared Task. In *Proceedings of CoNLL 2018: The SIGNLL Conference on Computational Natural Language Learning*, pages 197–207, Stroudsburg, PA, USA. Association for Computational Linguistics.

Milan Straka, Jana Straková, and Jan Hajic. 2019. UDPipe at SIGMORPHON 2019: Contextualized Embeddings, Regularization with Morphological Categories, Corpora Merging. In *Proceedings of the 16th Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 95–103, Florence, Italy. Association for Computational Linguistics.

Milan Straka, Jana Straková, and Jan Hajič. 2019. Evaluating Contextualized Embeddings on 54 Languages in POS Tagging, Lemmatization and Dependency Parsing. *arXiv e-prints*, page arXiv:1908.07448.

Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–20, Brussels, Belgium. Association for Computational Linguistics.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019. AMR parsing as sequence-to-graph transduction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 80–94, Florence, Italy. Association for Computational Linguistics.