CUHK at MRP 2019: Transition-Based Parser with Cross-Framework Variable-Arity Resolve Action

Sunny Lai[1,4], Chun Hei Lo[2], Kwong Sak Leung[1,4] and Yee Leung[3,4]
[1]Department of Computer Science and Engineering
[2]Department of Systems Engineering and Engineering
[3]Department of Geography and Resource Management
[4]Institute of Future Cities
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{slai, ksleung}@cse.cuhk.edu.hk,
chlo@se.cuhk.edu.hk, yeeleung@cuhk.edu.hk

Abstract

This paper describes our system (RESOLVER) submitted to the CoNLL 2019 shared task on Cross-Framework Meaning Representation Parsing (MRP). Our system implements a transition-based parser with a directed acyclic graph (DAG) to tree preprocessor and a novel cross-framework variable-arity resolve action that generalizes over five different representations. Although we ranked low in the competition, we have shown the current limitations and potentials of including variable-arity action in MRP and concluded with directions for improvements in the future.

1 Introduction

This paper describes our submission[1] to the CoNLL 2019 shared task on Cross-Framework Meaning Representation Parsing (Oepen et al., 2019). The task requires participants to develop a unified system for parsing sentences under five different meaning representation frameworks, which are DELPH-IN MRS (DM; (Ivanova et al., 2012)), Prague Semantic Dependencies (PSD; (Hajic et al., 2012; Miyao et al., 2014)), Elementary Dependency Structures (EDS; (Oepen and Lønning, 2006)), Universal Conceptual Cognitive Annotation (UCCA; (Abend and Rappoport, 2013)) and Abstract Meaning Representation (AMR; (Banarescu et al., 2013)). Given a sentence together with its companion data (e.g. morpho-syntactic parse results) as input, the parser system should generate five graphs according to each frameworks' rules.

Transition-based approaches have been shown useful in parsing a spectrum of semantic graphs, including bi-lexical dependency graphs (flavor 0, e.g. DM, PSD), general anchored semantic graphs (flavor 1, e.g. EDS, UCCA), and unanchored semantic graphs (flavor 2, e.g. AMR). Previous transition-based parsing systems define a set of constant-arity transition actions[2] and these systems learn to select the best action at each state. Constant-arity parser actions work well for tackling individual tasks, but may not generalize well across representations because:

- The graph representation details are different across frameworks. i.e. the edge directions and labels are different when comparing figure 2a and 2c but they describe the same dependency in terms of semantics. The parser will have to learn two actions separately (LEFT-EDGE and RIGHT-EDGE) as the actions have different semantics depending on the framework used.
- Parsing actions can be unique for specific frameworks defined by different authors (Table 1). i.e. Action NODE(X) in UCCA creates a new node without node label, which may not be a suitable action for other frameworks.

As the primary focus of the task is about developing a robust model that unifies the learning process across different semantic graph banks, we develop our system following the traditional transition-based approach, while adding a DAG-to-Tree preprocessor and a set of cross-representation variable-arity actions in an attempt to tackle these two generalization problems. By converting graphs of all five frameworks to a common tree structure using the DAG-to-Tree prepro-

[1]Our submission is open-sourced in GitHub: https://github.com/Yermouth/mrp2019

[2]For instance, in basic arc-standard transition system (Nivre, 2008), SHIFT takes one node as argument and REDUCE takes two. The number of arguments (arity) for the action is constant and will not change depending on the word being parsed.

Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 CoNLL, pages 104–113
Hong Kong, November 3, 2019. ©2019 Association for Computational Linguistics
https://doi.org/10.18653/v1/K19-2010

MRP	F	Actions	Author
PSD	0	LEFT-REDUCE(L), RIGHT-SHIFT(L), NO-SHIFT, NO-REDUCE, LEFT-PASS(L), RIGHT-PASS(L), NO-PASS	(Wang et al., 2018)
UCCA	1	SHIFT, REDUCE, **NODE(X)**, LEFT-EDGE(X), RIGHT-EDGE(X), LEFT-REMOTE(X), RIGHT-REMOTE(X), SWAP, FINISH	(Hershcovich et al., 2017)
AMR	2	SHIFT, REDUCE, RIGHT-LABEL(R), LEFT-LABEL(R), SWAP, MERGE, PRED(N), **ENTITY(L)**, GEN(N)	(Guo and Lu, 2018)
*	*	SHIFT, IGNORE, RESOLVE	This paper

Table 1: Transition-based parsing actions defined by different authors.

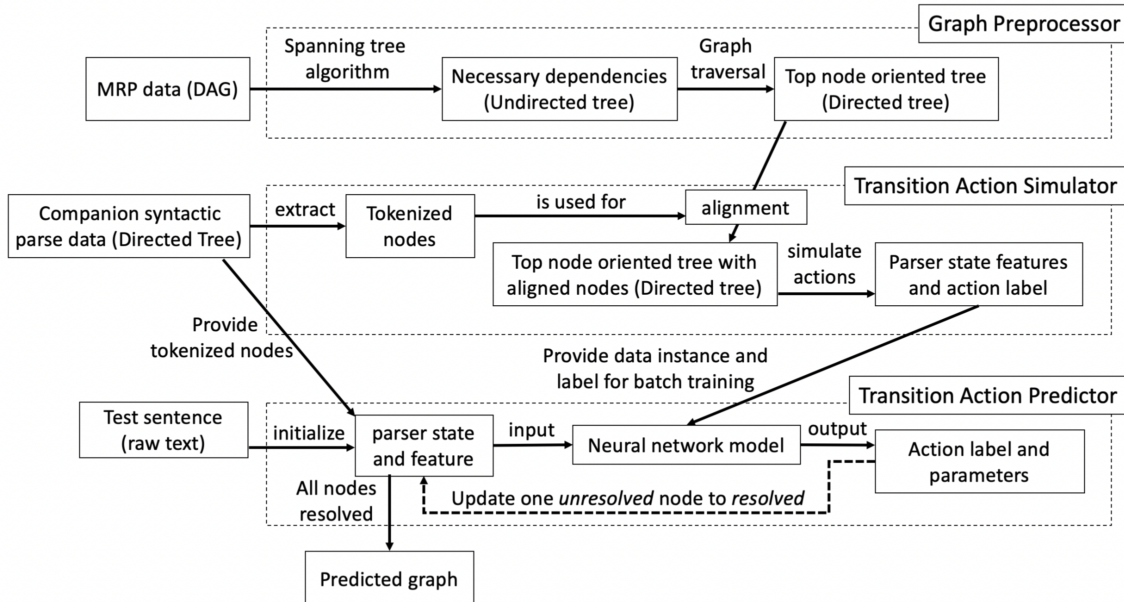


Figure 1: System pipeline diagram.

cessor, we can describe the tree generation process using three common high-level actions — SHIFT, IGNORE and RESOLVE.

The three actions in our system are most similar to the actions defined in the non-binary bottom-up shift-reduce constituent parsing strategy of Fernández-González and Gómez-Rodríguez (2018). SHIFT and IGNORE both have an arity of one. Unlike standard binary REDUCE action which handles the relationship between two nodes at a time, RESOLVE is a cross-framework variable-arity action that can reduce multiple nodes and resolve their dependency simultaneously. We introduce the RESOLVE action so that there is no need to include additional binarization of the dependencies and reduce the number of transitions as mentioned by Fernández-González and Gómez-Rodríguez. It is also more natural to consider the dependency of multiple nodes jointly as meaning representations like semantic frames usually involve multiple arguments.

The main difference between RESOLVER and the strategy of Fernández-González and Gómez-Rodríguez is that their strategy handles only constituent parsing problem while RESOLVER can handle cross-framework parsing problem. Our cross-framework RESOLVE action can be customized by generating framework-specific subgraphs.

Our submission ranked 13th overall in the post-evaluation period of the shared task. Although we ranked low in the task, we have experimented with adding variable-arity actions to the transition-based parsing approach and investigated its downsides. We studied why variable-arity transition actions are hard to learn and propose future directions for improving the system to predict variable-arity transition actions more accurately.

The rest of the paper is organized as follows: Section 2 describes the our system architecture. Section 3 details the model training steps. We analyze and discuss the result in Section 4 and conclude our work in Section 5.

2 System Architecture

Our system pipeline (Figure 1) is divided into three main components — DAG-to-Tree preprocessor, transition action simulator, and transition action predictor. First, we preprocess the meaning representation data and align it with the companion syntactic parse data to generate a top-node oriented tree structure. Then, we generate the transition actions required to reproduce the tree structure and extract the features involved in each action state. Finally, we train the neural network model to predict the correct actions.

2.1 DAG-to-Tree Preprocessor

Although the five frameworks differ in terms of the nodes and edges used, they are essentially conveying similar semantic messages. In an attempt to tackle the first generalization problem, our DAG-to-Tree preprocessor focuses on transforming the five frameworks into a common tree representation.

Our preprocessor converts directed acyclic graphs (DAGs) to top-node oriented tree structures. As the top-node of a sentence represents the most important message or word, they are similar amongst the five representations for the same sentence. Therefore, we can transform the five representations to a similar tree structure, where the root of the tree is the top-node.

As there are mature and standardized systems and algorithms for tackling tree-structured syntactic parsing, tree approximations schemes for transforming semantic dependency graphs to trees have been proposed (Schluter et al., 2014; Agić et al., 2015). While most of the proposed schemes are lossy, heuristics are applied to reduce information loss. For instance, the graph *packing* scheme (Schluter et al., 2014) use a set of 99.6%-reversible graph transformations to secure graph information, and the graph *deletion* scheme (Agić et al., 2015) remove minimum number of edges (worst case 5.7%) from undirected cycles in digraph to generate tree approximation.

2.1.1 Tree Approximation

Following the *deletion* scheme, we run an algorithm based on Kruskal's spanning tree algorithm (Kruskal, 1956) to select the edges for forming an undirected tree, and determine the edge direction of the edges in the tree by traversing the graph from top-node to every child recursively. The latter part is intuitive as the edge direction is unique (anti-arborescence) once the root of the undirected tree is fixed. As for graphs with more than one top node, we find the common ancestor of these top nodes and keep the graph if the ancestor is the root of the tree.

As for the undirected tree generation process, we first sort the nodes according to their appearance in the sentence, and assign the nodes with its appearance index in ascending order (i.e. Node anchored to the first word in the sentence have appearance index 1).

Then we extract the appearance index of the source node and the target node for each edge, and sort the edge in ascending order first by the maximum appearance index involved, and then by the minimum appearance index regardless of the edge direction (i.e. An edge with appearance indexes 1 and 3 will be placed in front of an edge with indexes 1 and 5).

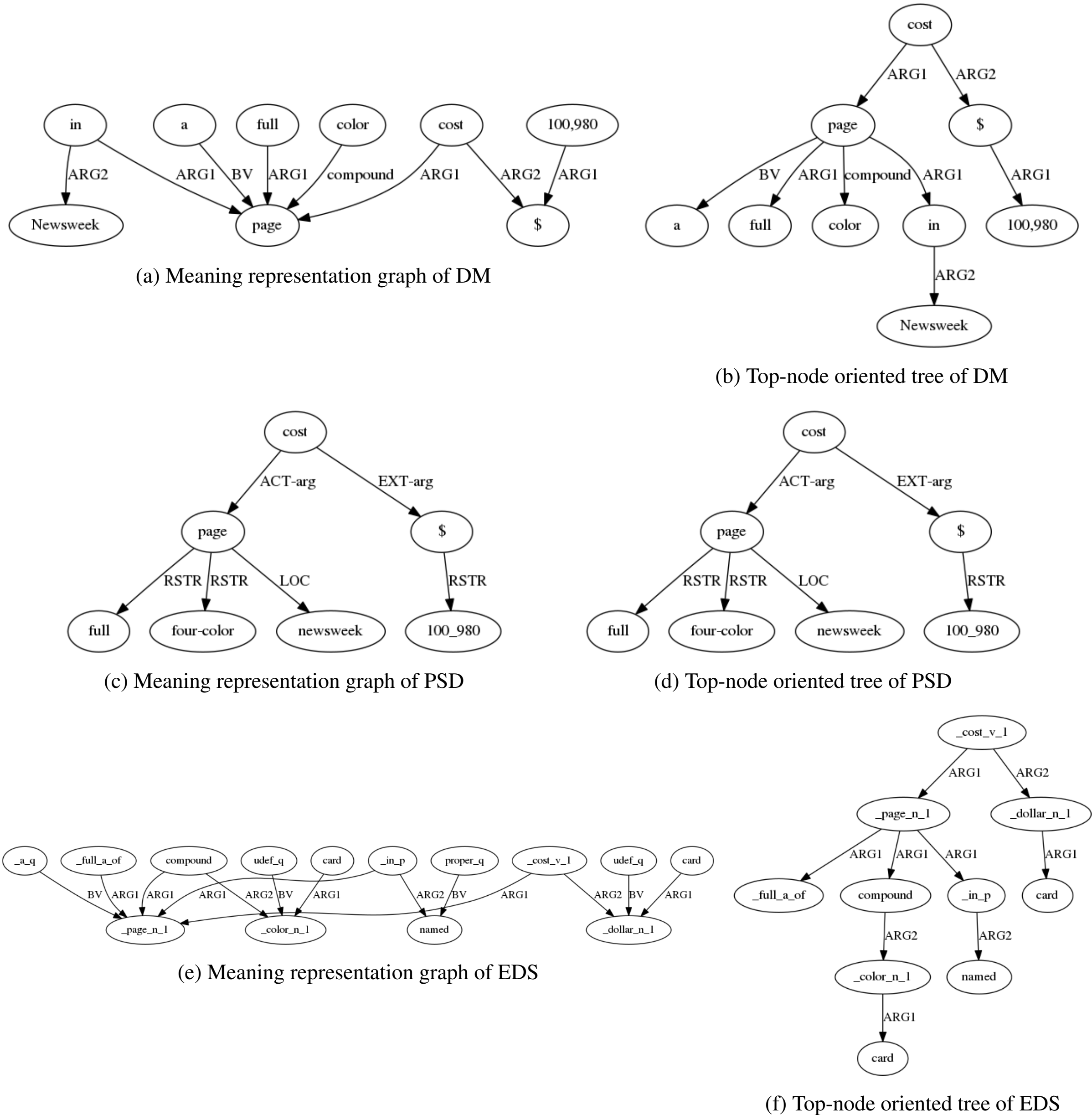
Finally, we initialize meaning representation nodes as forest in a graph, and add the sorted edge one by one to the graph if the edge connects to two different trees. After traversing the resulting graph from top-node, a set of edges accompanied with its direction is obtained and we refer to these edges as major edges (e.g. primary edges in UCCA). Other edges not in the major edge set are considered as minor edges. Minor edges can exist in PSD and UCCA, where one node can have multiple parents. For instance, nodes in UCCA can have a non-remote edge (major edge) with label "C" and a remote edge with label "A". For EDS specifically, edges that involve quantifiers are considered as minor edges at the moment to facilitate alignment.

In figure 2, 2a, 2c and 2e are the original meaning representation graphs and 2b, 2d, 2f are the top-node oriented trees created by using only the major edges after preprocessing. All three frameworks have the same top-node *"_cost_v_1"*.

Edge directions between the node *"page"* and its children are changed in figure 2b as *"cost"* is the top-node and traverse to node *"page"* before reaching nodes *"a"*, *"full"*, *"color"* and *"in"*.

Figure 2d is the same as 2c as the original graph is a tree and the edges' direction follow the traversal order from top-node.

As for figure 2f, minor edges including the edge with label *"BV"* from node *"udef_q"* to node *"_dollar_n_1"* are dropped in the current prepro-



(a) Meaning representation graph of DM

(b) Top-node oriented tree of DM

(c) Meaning representation graph of PSD

(d) Top-node oriented tree of PSD

(e) Meaning representation graph of EDS

(f) Top-node oriented tree of EDS

Figure 2: Meaning representation graphs of DM, PSD and EDS frameworks, accompanied with their top-node oriented tree after applying the DAG-to-Tree preprocessor for the sentence *"A full, four-color page in Newsweek will cost $100,980."*.

cessing procedures.

After these conversions, by comparing figure 2a, 2c, 2e with 2b, 2d, 2f, we can easily observe that the dependencies for the top-node oriented trees for are more unified as they are aligned with the top-node and its dependencies from the tree root. Despite the difference between DM, PSD and EDS in handling specific words (i.e. *"a"* is kept in DM and dropped in PSD), the general dependency structure is now more similar (i.e. all framework express that node *"page"* and *"$"* are necessary for resolving the complete semantics of the top-node *"cost"*).

2.1.2 Limitation

Limitations of the top-node oriented tree representation are apparent. The current representation sacrifices minor edges to retain the cross framework tree structure using the major edges. In this paper, we adopt the graph *deletion* scheme and mainly focus on tackling major edges that are common amongst the five frameworks. We leave minor edges and the use of graph *packing* scheme as future work.

2.2 Transition Action Simulator

To solve the second generalization problem, we define three actions: SHIFT, IGNORE and RESOLVE as the high-level actions in our action set which is common amongst the five frameworks. The tokenized nodes provided by the morpho-syntactic parse tree are the basic units for applying the actions. We initialize the parser state with a queue that stores all the tokenized nodes and an empty stack that stores the processed tokenized nodes.

2.2.1 Shift and Ignore

SHIFT and IGNORE are two constant-arity actions identical for all representations, and both apply directly to the first tokenized nodes in the queue. While both actions pop the first tokenized node from the token queue, SHIFT pushes the popped node to the stack and sets its state to unresolved, while IGNORE omits the popped node and move on to the next tokenized node in the queue. This action is required as the tokenization method of the syntactic parse is different from that of the MRP. Tokenized nodes in the syntactic parse can be ignored by the representation, for instance, verbs like "is" are omitted by DM, while it is preserved in PSD. From our observation, whether the word is ignored or not depends on only itself but not its neighbor nodes, so we can apply the action directly to the queue without considering the state of the stack.

2.2.2 Resolve

RESOLVE is a variable-arity and representation-customizable action. This action is similar to LEFT-REDUCE and RIGHT-REDUCE, but instead of reducing only 2 nodes at each time, RESOLVE can reduce an arbitrary number of nodes in one single action. We required our system to learn the dependencies of multiple nodes jointly in order to determine frame information in a holistic manner.

This action is mainly parameterized by n (arity), the number of nodes from the top of the stack to be reduced (n is a strictly positive integer). The first n nodes must include one and only one *unresolved* node (i.e. the most recently pushed *unresolved* node in the stack). After an *unresolved* node is resolved, it is pushed into the stack. As we have obtained a top-node oriented tree representation from the DAG-to-Tree preprocessor, the dependencies of each node of the tree are defined explicitly and RESOLVE is applied when an *unresolved* node's children are all *resolved*. For instance, in Figure 2(b), the top-node of the graph is *"cost"* and its dependencies is *"page"* and *"$"*. To RESOLVE the node *"cost"*, we need to first RESOLVE both *"page"* and *"$"*, which further depends on their own children. The number of reduced node n in this case is 3 (2 resolved nodes *"page"* and *"$"* plus 1 unresolved node *"cost"*).[3] If a node is a leaf node, n in this case would be 1 as only one node is involved.

After selecting n nodes from the stack, the RESOLVE action build the edges between the *resolved* nodes and the *unresolved* ones, and give node label and properties for the *unresolved* node. Finally, the resolved node is pushed back to the stack.

2.3 Alignment

Aligning a sentence S to a graph $G = \langle V, E \rangle$ of meaning representation gives a mapping between the tokens of S and V. Formally, given a parse tree of S with tokenized nodes $\langle N_0, N_1, \ldots, N_n \rangle$, with each N_i containing $\langle a_{start}, a_{end} \rangle$ of S: pair of from-to sub-string indices, pos: part of speech tag, and $lemma$: lemmatized form, we aim to produce an alignment $V = \langle M_0, M_1, \ldots, M_m \rangle$, where each node object M_i contains $\langle a_{start}, a_{end} \rangle$: pair of from-to sub-string indices to S, pos: part of speech tag, $frame$: semantic frame (optional) and $label$: node label (Figure 3).

As the alignment of the tokenized nodes in the companion parse to the nodes in the meaning representation graph is not given, we devised alignment strategies for the respective framework using anchors and parse information. For DM and PSD, an oracle look-ahead algorithm is designed, where the alignment is conducted as guided by a set of heuristic rules manually derived from the train data. For each sentence, the alignment process proceeds by scanning tokenized nodes of the parse tree from left to right, one at a time. Each node is either ignored or aligned to one node of the meaning representations.

For DM, as white-listed resources are provided, we allow more aggressive grouping and prediction on semantic frames. Generally, $M_j.pos$ and $M_j.label$ will be copied directly from the corresponding $N_i.pos$ and $N_i.lemma$ respectively, with a few exceptions handled the other ways; and $M_j.frame$ are predicted using a simple count-based approach with train data. Multi-word ex-

[3]This corresponds to the last RESOLVE action in Table 1

Action	n	Stack	Tokenized Node Queue	RESOLVE Details
		[]	[A, full, ...]	
SHIFT		[A]	[full, ,, ...]	
RESOLVE	1	[**a**]	[full, ,, ...]	Leaf node
SHIFT		[a, full]	[,, four-color, ...]	
RESOLVE	1	[a, **full**]	[,, four-color, ...]	Leaf node
IGNORE		[a, full]	[four-color, page, ...]	
SHIFT		[a, full, four-color]	[page, in, ...]	
RESOLVE	1	[a, full, **color**]	[page, in, ...]	Leaf node
SHIFT		[a, full, color, page]	[in, Newsweek, ...]	
SHIFT		[a, full, color, page, in]	[Newsweek, *will*, ...]	
SHIFT		[a, full, color, page, in, Newsweek]	[*will*, cost, ...]	
RESOLVE	1	[a, full, color, page, in, **Newsweek**]	[*will*, cost, ...]	Leaf node
RESOLVE	2	[a, full, color, page, **in**]	[*will*, cost, ...]	**in** $\xrightarrow{\text{ARG2}}$ Newsweek
RESOLVE	5	[**page**]	[*will*, cost, ...]	**page** $\xrightarrow{\text{BV}}$ a, **page** $\xrightarrow{\text{ARG1}}$ full **page** $\xrightarrow{\text{compound}}$ color, **page** $\xrightarrow{\text{ARG1}}$ in
IGNORE		[page]	[cost, $, ...]	
SHIFT		[page, cost]	[$, 100,980]	
SHIFT		[page, cost, $]	[100,980]	
SHIFT		[page, cost, $, 100,980]	[]	
RESOLVE	1	[page, cost, $, **100,980**]	[]	Leaf node
RESOLVE	2	[page, cost, **$**]	[]	**$** $\xrightarrow{\text{ARG1}}$ 100,980
RESOLVE	3	[**cost**]	[]	**cost** $\xrightarrow{\text{ARG1}}$ page, **cost** $\xrightarrow{\text{ARG2}}$ $

Initial tokenized nodes queue: [A, full, ,, four-color, page, in, Newsweek, *will*, cost, $, 100,980]

Table 2: Actions required to generate the Figure 2(b) graph for the sentence *"A full, four-color page in Newsweek will cost $100,980."*. The column n indicates the number of nodes to be resolved. When $n = 1$, the resolved node is a leaf node. When $n > 1$, the column RESOLVE details shows the edge involved in the RESOLVE process. Resolved nodes are in normal font. Unresolved nodes are underlined, and the nodes to be resolved in each action are denoted in boldface. The number of RESOLVE in the actions is the same as the number of nodes in the top-node oriented tree. The two IGNORE actions ignore the tokenized nodes *","* and *"will"* respectively.

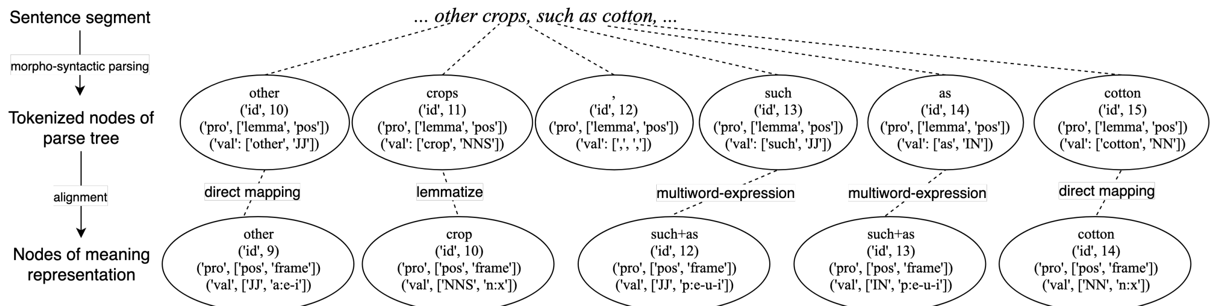
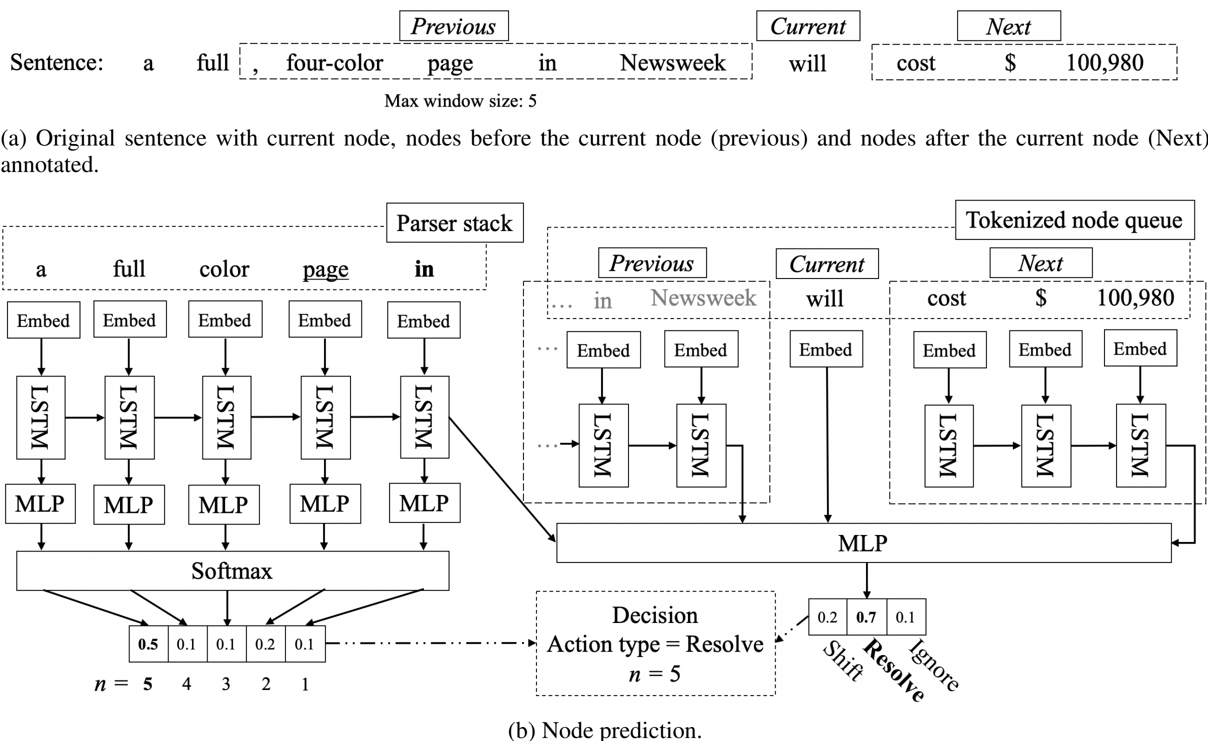


Figure 3: Example of alignment of nodes of DM meaning representation.

pressions (MWE) are also accounted for during the alignment through a greedy look-ahead mechanism, i.e. searching for MWE in S that appeared in train data or the SDP 2016 data (Oepen et al., 2016), which is one of the white-listed resources for the task. Figure 3 illustrates the alignment process from tokenized nodes to nodes of DM representation: MWE "*such as*" is handled with heuristics to produce two nodes; "*crops*" is lemmatized as the label of the produced node; Frames are copied except for punctuation ",", which is ignored. Details of the alignment process are provided in the supplementary material.

For PSD, only frames that appeared in train data were inferred. Similar to the approach for DM, alignment is generally done by copying $M_j.pos$ and $M_j.label$ from the corresponding $N_i.pos$ and $N_i.lemma$ respectively; and $M_j.frame$ are predicted only for verbs using the same count-based approach as for DM. Multi-word expressions are also accounted for during the alignment process through a greedy look-ahead mechanism. PSD also includes the use of non-lexical nodes for abstract concepts (e.g. *#perspron* for personal pronoun), and they are aligned to N_i first, if possible, followed by lexical nodes.



(a) Original sentence with current node, nodes before the current node (previous) and nodes after the current node (Next) annotated.

(b) Node prediction.

Figure 4: Neural network architecture diagram of Action type prediction.

For both DM and PSD, given the tokens, frame predictions are done by a simple count-based method, i.e. we choose the most-occurred frame as in the train data given each token; if no such token is found in train data, we choose the first frame from the frame inventories of DM and PSD (white-listed resources) for the corresponding token or lemma. More robust statistical methods for frame prediction are left for future work.

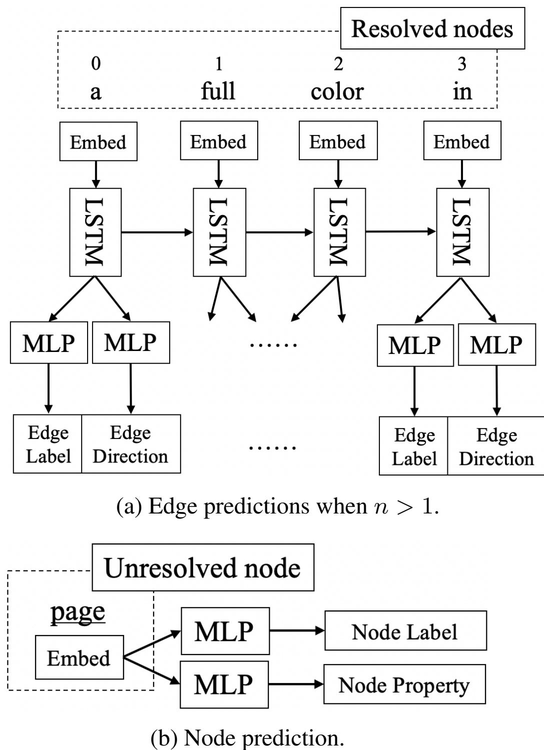
For EDS and UCCA, we use exact matching policy to match the anchors of the tokenized with the graph nodes. If one tokenized node is mapped to multiple graph nodes, we drop the whole graph in the current system. For AMR, we use the JAMR (Flanigan et al., 2014) alignment provided in the companion data to align the unanchored nodes to the tokenized nodes.

2.4 Neural Network Model

To determine the correct action for a particular parser state, we use two neural network models to first decide what action should be taken, and determine the framework details if the action is RESOLVE.

2.4.1 Action Type Prediction

Figure 4 describes the neural network architecture for predicting the actions. The nodes in the parser stack and tokenized node queue are first mapped to feature embeddings. The feature embedding of each node is created by concatenating the GloVe (Pennington et al., 2014) word embedding together with three randomly initialized embeddings for the features *word lemma*, *upos* and *xpos* provided by the syntactic parse. Then, we use LSTM (Hochreiter and Schmidhuber, 1997) layers to encode three nodes sequences: (1) nodes in the parser stack, (2) nodes before the current node and (3) nodes after the current node. For sequence (2) and (3) we limit the size of the sequence to be 5. We concatenate the hidden state at the last time step of the three sequences with the current node's feature embedding and feed it to a multilayer perceptron (MLP) to predict the action type. As we need n, the number of nodes to be reduced for the reduce action, we use the hidden states for every time step of sequence (1) and pass them to the same MLP, and then the softmax layer to predict the value of n. We choose the action type and n with the greatest probability to execute. If RESOLVE is to be executed, we extract the first n nodes from the parser stack, and proceed with the RESOLVE prediction.



(a) Edge predictions when $n > 1$.

(b) Node prediction.

Figure 5: Neural network architecture diagram of RESOLVE prediction.

2.4.2 Resolve Prediction

Figure 5 pictures the neural network architecture for predicting the label and properties of the nodes and edges in the RESOLVE process. If a leaf node is to be resolved ($n = 1$), then no edge is involved. we use the feature embedding of the unresolved node as input, and pass it to feature specific MLP for predicting the node label and properties. If more than one node is involved ($n > 1$), then we, in addition, predict the edge information by passing the feature embedding to an LSTM layer, followed by feature-specific MLPs for predicting edge label and directions.

2.4.3 Multi-Task Learning

To enable multi-task learning, we use the same neural network model for parsing all five frameworks. We shared the parameters of word embeddings and LSTM layers across frameworks, and separate the MLP parameters for each framework.

3 Training

3.1 Data

We use the official dataset as the development set to train our system. We use the DAG-to-Tree preprocessor and action simulator to generate action snapshots of the parser state features (parser stack and tokenized node queue) and action labels for each action applied, acting as the data instances for training the neural network model. A total of 169,780 MRP-parse data pairs are given, for which we generate 2,434,026 action snapshots as training data instances. Our system is required to predict the MRP graphs for 13,206 unseen sentences.

3.2 Implementation Details

Our system is packaged as an AllenNLP library (Gardner et al., 2017), which comprises DAG-to-Tree preprocessors, dataset readers, training instance iterators, neural network models and MRP graph predictors. The neural network model is implemented using Pytorch and support training with either CPU or single GPU setting. Time required for each procedure is summarized in table 3.

Procedures	**Required Time (hour)**
Run DAG-to-Tree preprocessor and action simulator using training data	10
Use AllenNLP data reader to read data instances	1.5
Train the neural network model (single GPU setting)	30 in total (2 per epoch)
Predict the MRP graph of testing data	8
Total	49.5

Table 3: Running time for each procedure.

3.3 Batch training

As each graph is broken down into training instances for each action and the size of the instances is large, batch training is necessary to speed up the training process. We group the data instance into mini-batch of size 100 by their prediction type (whether it is action type prediction or resolve prediction), meaning representation framework, and the length of the stack and queue to facilitate batch training. Both training batches and training instances in the same framework batch are shuffled in each epoch.

4 Results and Discussion

4.1 Official Results

According to the results announced, we ranked 13th overall in the post-evaluation period of the shared task. We compared the results of our system with a similar transition-based parser TUPA (Hershcovich and Arviv, 2019) in Table 4. Our

Submissions	tops			labels			properties			anchors			edges		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
TUPA(multi)	0.67	0.57	0.616	0.40	0.55	0.457	0.29	0.42	0.327	0.68	0.60	0.626	0.30	0.45	0.347
RESOLVER	0.51	0.50	0.502	0.34	0.40	0.365	0.29	0.35	0.317	0.55	0.59	0.568	0.10	0.10	0.095

Submissions	attributes			all		
	P	R	F	P	R	F
TUPA(multi)	0.06	0.03	0.037	0.39	0.57	0.453
RESOLVER	0.00	0.00	0.00	0.36	0.41	0.378

Table 4: Final results of our system compared with the transition based parser TUPA. All scores are calculated according to the MRP metric.

system performs slightly worse than TUPA in general, while we performed much worse in the edges component.

4.2 Discussion

We analyze our system and investigate three reasons for causing the low performance.

- Variable-arity actions are hard to learn. Our system predicts the action type with accuracy around 0.8 across frameworks, but cannot predict the number of nodes, i.e. n, to be reduced well (less than 0.35). As the number of training instances with $n = 1$ is much larger than that of $n > 1$, we believe the unbalanced number of training examples can be a hindrance for learning to predict n correctly.
- Information loss happens when converting graphs to tree structures. As we are using the DAG-to-Tree preprocessor to convert graphs to top-node oriented trees using major edges, we ignore minor edges in the current model and loss features for predicting the action and chances for predicting them. Moreover, we cannot find direct and empirical proof of why this top-node oriented tree conversion can help the parsing process.
- Model design can still be improved. There are numerous variations including neural network architecture, hyperparameters, action set, feature set, etc, that our team can experiment with under the variable-arity transition action and top-node oriented tree paradigm. More time is required to test if this is a valid approach to tackle the parsing problem in general.

5 Conclusion

We present RESOLVER, the first transition-based parser with top-node oriented DAG-to-Tree preprocessor and variable-arity actions to the best of our knowledge. We aim to create a generalized representation and parsing steps of the five graphs. We discuss the benefits and limitations of adding variable-arity actions, and we will continue to work on our system to show the practical usefulness of allowing variable-arity transition actions in transition-based meaning representation parsers.

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.

Željko Agić, Alexander Koller, and Stephan Oepen. 2015. Semantic dependency graph parsing using tree approximations. In *Proceedings of the 11th International Conference on Computational Semantics*, pages 217–227.

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.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2018. Faster shift-reduce constituent parsing with a non-binary, bottom-up strategy. *arXiv preprint arXiv:1804.07961*.

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1426–1436.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. Allennlp: A deep semantic natural language processing platform.

Zhijiang Guo and Wei Lu. 2018. Better transition-based amr parsing with a refined search space. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1712–1722.

Jan Hajic, Eva Hajicová, Jarmila Panevová, Petr Sgall, Ondrej Bojar, Silvie Cinková, Eva Fucíková, Marie Mikulová, Petr Pajas, Jan Popelka, et al. 2012. Announcing prague czech-english dependency treebank 2.0. In *LREC*, pages 3153–3160.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A transition-based directed acyclic graph parser for ucca. *arXiv preprint arXiv:1704.00552*.

Daniel Hershcovich and Ofir Arviv. 2019. TUPA at MRP 2019: A multi-task baseline system. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*, pages 28 – 39, Hong Kong, China.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

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*, pages 2–11. Association for Computational Linguistics.

Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

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 (SemEval 2014)*, pages 335–340.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Stephan Oepen, Omri Abend, Jan Hajič, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, Jayeol Chun, Milan Straka, and Zdeňka Urešová. 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, China.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinková, Dan Flickinger, Jan Hajic, Angelina Ivanova, and Zdenka Uresova. 2016. Towards comparability of linguistic graph banks for semantic parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, pages 3991–3995.

Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-based mrs banking. In *LREC*, pages 1250–1255.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Natalie Schluter, Anders Søgaard, Jakob Elming, Dirk Hovy, Barbara Plank, Hector Martinez Alonso, Anders Johanssen, and Sigrid Klerke. 2014. Copenhagen-malmö: Tree approximations of semantic parsing problems. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 213–217.

Yuxuan Wang, Wanxiang Che, Jiang Guo, and Ting Liu. 2018. A neural transition-based approach for semantic dependency graph parsing. In *Thirty-Second AAAI Conference on Artificial Intelligence*.