Transition-Based Meaning Representation Parsing

Literature Review

Chase Ting Chong
University of Cape Town
Cape Town, South Africa
tngcha001@myuct.ac.za

ABSTRACT

Meaning Representation Parsing (MRP) is the task of encoding a sentence into a meaning representation. We provide an overview of how this meaning representation is represented in the Elementary Dependency Structures (EDS) format. Following this is an overview of neural network architectures and transition-based parsers. Finally we compare various state of the art parsers to the current state of the art transition-based EDS parser to find gaps in the literature. We conclude that transition-based EDS parsers are outperformed by transformer based parsers and transformers have yet to be applied to EDS parsers beyond deep contextualised words representations.

KEYWORDS

Natural Language Processing, Natural Language Understanding, Meaning Representation Parsing, Semantic Graph Parsing, Neural Networks

1 INTRODUCTION

Meaning Representation Parsing (MRP) is the encoding of sentences into *meaning representations* [1]. These meaning representations take the form of a *semantic graph*, there are many existing meaning representation frameworks which differ in their formal and linguistic assumptions [19, 20]. Elementary Dependency Structures (EDS) is one of these frameworks that adapts Minimal Recursion Semantics (MRS) into variable free semantic graphs [21].

There are many different approaches to MRP of which transition-based MRP is one of the most researched in the literature [19, 20]. Tranisition-based MRP is adapted from transition-based dependency parsing which is the task of deriving a *syntactic dependency graph* from a sentence [16]. This is achieved by predicting a sequence of transition actions via a neural network [5].

In this paper we aim to provide an overview of EDS transition-based dependency concepts and commonly used neural network architectures. We will then analyse the state of the art transition-based parsers as well as state of the art EDS parsers in an attempt to find a gap in EDS transition-based MRP.

2 MEANING REPRESENTATION

2.1 Trees versus Graphs

In the context of Dependency Parsing (§4) and MRP (§5) the output of both parsing types can be represented as rooted, labelled, connected, directed graphs [12, 16]. These graphs are called Dependency Graphs and Semantic Graphs respectively.

Dependency graphs further restrict the structure to that of a *tree* [16]. Trees place the additional restrictions in that they are acyclic and do not contain *reetrancies* [12, 24]. A reentrancy is a node that contains in-degree of 2 or higher [12].

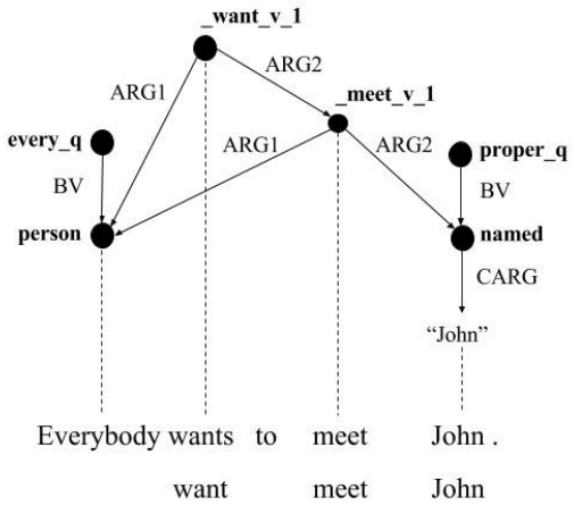

Figure 1: Semantic representation of the sentence "Everybody wants to meet John" in the EDS graph format [3].

Semantic graphs are more general, allowing them to capture deeper semantic meaning as opposed to tree structures that can only capture shallow syntax [24]. Trees do however have desirable computational and linguistic properties [24], therefore parsing to semantic graphs requires a more complex approach (§5).

2.2 Elementary Dependency Structures

Elementary Dependency Structures (EDS; [21]) converts Minimal Recursion Semantics (MRS; [6]) to a semantic graph. MRS is a framework for computational semantics that can be used for parsing or generation [6]. A property of MRS is that is *underspecified* i.e. it does not resolve scope ambiguity [6]. This allows multiple scope-resolved logical representations to be derived from a single MRS structure [3]. The conversion of MRS to a semantic graph discards this property [3, 20].

Dependency graphs have an ordered one-to-one mapping of their nodes to the sentence used to construct the graph, i.e. the first node in the graph corresponds to the first word of the input sentence, the second node to the second word etc. [16, 20]. This is however, not the case with EDS semantic graphs. EDS semantic graphs have a many-to-many mapping this means that any number of nodes could correspond to any substring from the input sentence

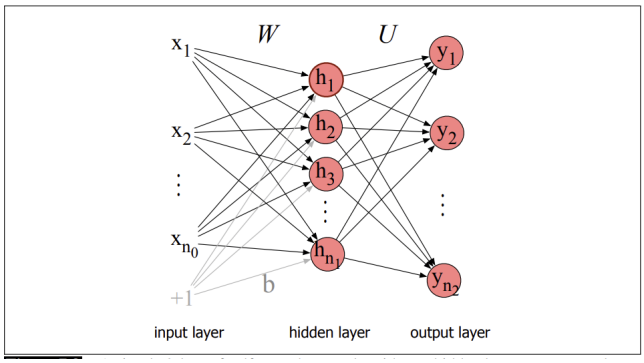

Figure 2: A simple 2-layer feedforward network [11]

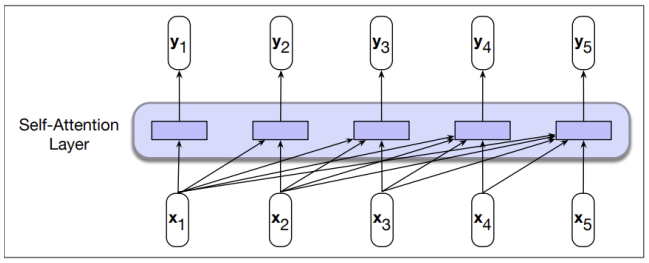

Figure 3: A self attention layer in a transformer [11]

[20]. This mapping of nodes to substrings is called alignment or anchoring [20].

Figure 1 shows an EDS graph. Node labels are called *predicates* and edge labels *arugments*.

3 NEURAL NETWORKS

The simplest kind of neural network is a *feedforward* neural network, a multi-layer network that does not contain cycles [11]. A feedforward neural network consists of an input layer, any number of hidden layers, and an output layer. Figure 2 shows a simple 2-layer feedfoward network. The input layer is represented by an input vector $\mathbf{x}$ containing each input x_i. The input vector is multiplied by a weight matrix $\mathbf{W}$ and an additional bias term b is added the result of this is a vector that is used as input to the hidden layer. The hidden layer then applies a non-linear function called the *activation* function σ to this vector returning an output vector $\mathbf{h}$. $\mathbf{h}$ is then multiplied by another weight matrix $\mathbf{U}$ resulting in a vector $\mathbf{z}$. $\mathbf{z}$ is then used as input to the output layer which then applies a normalizing function, typically *softmax* which converts the real number vector $\mathbf{z}$ into a vector $\mathbf{y}$ that encodes a probability distribution. This is summarized by the below equations. [11]

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\mathbf{y} = softmax(\mathbf{z})$$

Training a neural network entails finding values for the weight matrices and biases the details of which we will not go into in this paper.

3.1 Long Short Term Memory

Long Short Term Memory (LSTM) is an extension to Recurrent Neural Network (RNN). A RNN is any network that contains a cycle [11]. A simple RNN processes input in time steps and at each time step uses information from the previous time step's hidden nodes h_{t-1} in the current hidden node h_t calculation.

RNNs have two problems that LSTM tries to solve. Firstly when processing a sequence the information encoded in the current hidden state h_t is mostly based on recent inputs from previous hidden states i.e. h_{t-1} therefore inputs that were processed long before the current time step does not have a strong influence on current decisions, which is an important feature in many language application. Secondly training RNNs is very difficult as hidden nodes are dependent on previous hidden nodes. [11]

LSTM attempts to solve these problems by adding an extra memory cell that stores the context c_t. The model now sends this context along with the hidden state at each time step. The context is controlled by three gates [11]:

- The **forget** gate deletes information from the context that is no longer needed.
- The **add** gate decides what information should be added to the current context.
- The **output** gate decides which information is relavant for the current hidden node.

3.2 Transformers

A Transformer maps sequences input vectors $(x_1, ..., x_n)$ to sequences $(y_1, ..., y_m)$ of output vectors and are made up of transformer blocks, multi-layer networks made by combining simple linear layers, feedforward layers, and self-attention layers [11].

What differentiates transformers from other models is it's *self-attention* mechanism. The idea behind attention is to compare a collection of items based on their relevance to the current context. Self-attention is simply the comparing of an item in a collection to other items in the collection. For example in Figure 3 the computation of y_4 is based on which elements x_1 to x_4 are most relevant. [11]

There may be various ways to classify something as relevant to a context, to address this transformers use *multihead self-attention layers* also known as *multihead attention*. This is simply a set of self-attention layers, each layer called a *head*, now each head attend to a different representation of relevance. [11]

Since transformers do not use recurrence they do not encounter the problems that RNN face. They are superior in quality, easily parallelizable and require significantly less time to train. [25].

4 TRANSITION-BASED DEPENDENCY PARSING

The goal of dependency parsing is to parse a sentence into a dependency graph [15, 16]. A dependency graph of a sentence represents each word and its syntactic modifiers through labeled directed arcs, as shown in Figure 4 [15]. For an arc from node i to node j, i is called the *head* and j the *dependent* [17].

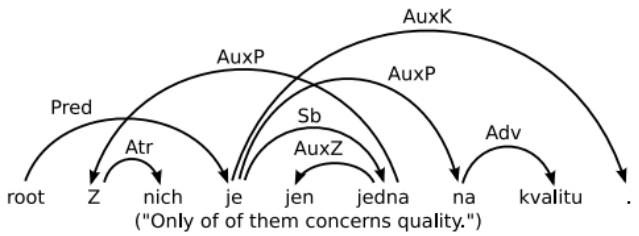

Figure 4: An example dependency graph [15]

Transition	Configuration			
	([0],	[1, . . . , 9],	∅	)
SHIFT ⟹	([0, 1],	[2, . . . , 9],	∅	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0],	[2, . . . , 9],	$A_1 = \{(2, \text{NMOD}, 1)\}$	)
SHIFT ⟹	([0, 2],	[3, . . . , 9],	A_1	)
LEFT-ARC$_{\text{SBJ}}$ ⟹	([0],	[3, . . . , 9],	$A_2 = A_1 \cup \{(3, \text{SBJ}, 2)\}$	)
RIGHT-ARC$^e_{\text{ROOT}}$ ⟹	([0, 3],	[4, . . . , 9],	$A_3 = A_2 \cup \{(0, \text{ROOT}, 3)\}$	)
SHIFT ⟹	([0, 3, 4],	[5, . . . , 9],	A_3	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, 3],	[5, . . . , 9],	$A_4 = A_3 \cup \{(5, \text{NMOD}, 4)\}$	)
RIGHT-ARC$^e_{\text{OBJ}}$ ⟹	([0, 3, 5],	[6, . . . , 9],	$A_5 = A_4 \cup \{(3, \text{OBJ}, 5)\}$	)
RIGHT-ARC$^e_{\text{NMOD}}$ ⟹	([0, . . . , 6],	[7, 8, 9],	$A_6 = A_5 \cup \{(5, \text{NMOD}, 6)\}$	)
SHIFT ⟹	([0, . . . , 7],	[8, 9],	A_6	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, . . . 6],	[8, 9],	$A_7 = A_6 \cup \{(8, \text{NMOD}, 7)\}$	)
RIGHT-ARC$^e_{\text{PMOD}}$ ⟹	([0, . . . , 8],	[9],	$A_8 = A_7 \cup \{(6, \text{PMOD}, 8)\}$	)
REDUCE ⟹	([0, . . . , 6],	[9],	A_8	)
REDUCE ⟹	([0, 3, 5],	[9],	A_8	)
REDUCE ⟹	([0, 3],	[9],	A_8	)
RIGHT-ARC$^e_{\text{P}}$ ⟹	([0, 3, 9],	[ ],	$A_9 = A_8 \cup \{(3, \text{P}, 9)\}$	)

Figure 5: Arc Eager transition sequence for the sentence "Economic news had little effect on financial markets". Configurations are represented by a triple (stack, buffer, arcs) [17]

Transition-based dependency parsers construct dependency graphs by predicting a sequence of actions that create the graph via a *stack-based transition system* [16, 17]. Actions are predicted via a neural network based classifier that is trained by an oracle §4.4 [5]. Data-driven dependency models can be implemented very efficiently, an advantage of transition-based parsing in particular, is its linear run time [8, 9, 18]. We show two algorithms for transition-based parsing using stack-based transition systems (§4.2, §4.3)

4.1 Stack-Based Transition System

A stack-based transition system consists of parser configurations, a transition set, a way to map an input sentence to a starting configuration and set of configurations that when transitioned to terminates the algorithm [17].

- A parser configuration consists of a stack that holds nodes currently being processed, a buffer holding tokens (words in the sentence) that still need to be processed and a set of arcs.
- A transition set is a set of actions that perform a transformation to a parser configuration resulting in a new configuration.
- The initial configuration starts with an artificial root node 0 on the stack, nodes i, for every word w_i in the input sentence, on the buffer and the set of arcs is empty.
- A terminal configuration is any configuration where the buffer is empty.

4.2 Arc Standard Algorithm

The arc standard algorithm's transition set consists of three types of transitions [17]:

- **Left-Arc**$_l$ adds a dependency arc with label l from the node i at the top of the stack to the first node j in the buffer and then pops i off the stack, given that i is not the root node.
- **Right-Arc**$_l$ adds a dependency arc with label l from the first node j in the buffer to the node i at the top of the stack and and then pops i off the stack and replaces j with i at the front of the buffer, given that j does not already have a head.
- **Shift** removes the first node i in buffer and pushes it to the top of the stack.

The arc-standard parser is similar to the shift-reduce parser for context-free grammars. Where the left and right arc transitions correspond to reduce and shift to shift [17].

4.3 Arc Eager Algorithm

The arc eager algorithm's transition set consists of four types of transitions [17]:

- **Left-Arc**$_l$ adds a dependency arc with label l from the node i at the top of the stack to the first node j in the buffer and then pops i off the stack, given that i is not the root node and does not have a head.
- **Right-Arc**$_l$ adds a dependency arc with label l from the first node j in the buffer to the node i at the top of the stack and and then pops i off the stack and shifts j onto the stack, given that j does not already have a head.
- **Reduce** Pops the top node i from the stack, given that i has a head.
- **Shift** removes the first node i in buffer and pushes it to the top of the stack.

The arc eager algorithm tries to attach right dependents as soon as possible therefore the head and dependent must be stored on the stack until the dependent gets all its right dependents and can only then be popped of the stack using the reduce transition [17]. Figure 5 shows a example transition sequence.

4.4 Oracles

A core part of training classifiers for transition-based dependency parsers is an *oracle* which, when given a gold parse tree returns the optimal transition sequence [10].

4.4.1 Static Oracles. translate a given tree into a sequence of transitions that when run in sequence will return the gold tree. It is possible that multiple sequences exist that lead to the same gold tree, in this case static oracles define a forced derivation order resulting in a single sequence being returned. Static oracles are typically defined as rules for a given parser configuration. For example given a configuration with properties X and gold tree Y, then the correct transition is Z [10].

There are a few limitations to using static oracles. When there are multiple possible sequences, the sequence given by the oracle (because of the forced derivation order) may not be the easiest to

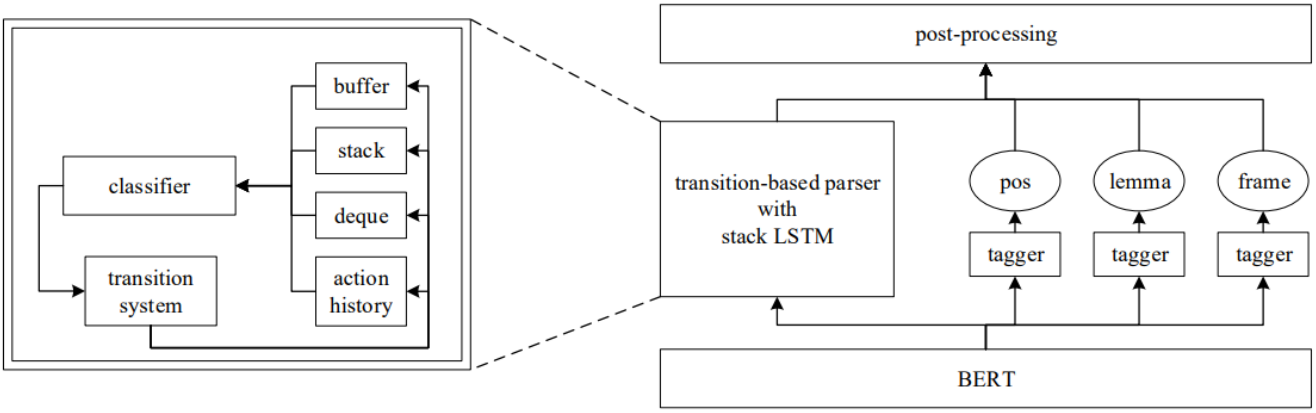

Figure 6: HIT-SCIR 2019 Architecture [4]

learn, one of the alternate sequences may give better results. In addition since the parsing is greedy it possible for the parser to deviate from the gold sequence to configurations where the correct tree is no longer reachable. The oracle does not have a mechanism to deal with these configurations, this leads to error-propagation as the parser's classifier is faced with configurations not observed in training [10].

4.4.2 Dynamic Oracles. check if a transition Z is valid in a configuration X when trying to create the best possible tree Y. Dynamic oracles do not restrict parsers to a single sequence but instead return all valid sequences leading to the gold tree. Another improvement of dynamic oracles is that, unlike their static counterpart, they are well-defined and correct for all configurations. For configurations that have deviated from the gold sequence, the dynamic oracle allows all transitions that lead to a tree with minimum loss compared to the gold tree thus mitigating error propagation [10].

5 TRANSITION-BASED MRP

5.1 General Graph Parsing

To extend transition-based dependency parsers to parse general graph structures the following transitions are added:

5.1.1 Supporting Reentrancies. To support **reentrancies** [24] presents a data-driven approach to parsing to directed acyclic graphs (DAGs) based on shift-reduce dependency parsing. To allow words to have multiple heads, two new parser transitions are added that do not remove nodes from the stack, thereby removing them from further consideration.

- **Left-Attach** creates a left arc between the top two nodes on the stack, provided that a right arc does not exist between the two nodes.
- **Right-Attach** creates a right arc between the top two nodes on the stack, provided that a left arc does not exist between the two nodes.

5.1.2 Supporting Crossing Edges. Traditional transition-based dependency parser produce *projective* graphs [16]. A projective graph is a graph that does not contain crossing edges when drawn in the halfplane above the sentence [12]. Figure 4 is a non-projective graph as it contains crossing edges.

To support **Crossing Edges** [18] presents a transition system for parsing non-projective trees. This is achieved by adding a new transition **Swap**. Swap moves the second node on the stack back to the buffer reversing the order of the top two nodes. A swap can only be performed when the top two nodes are in their original word order, this prevents nodes from being swapped more than once, and when the second node is not the root node. Reversing the order of the top two nodes allows **Left-Arc** and **Right-Arc** to create crossing edges therefore creating a non-projective graph.

5.2 Deep Contextualized Word Embeddings

One of the disadvantages of transition-based parsers is that because of its local greedy nature it is prone to search errors in sentences that require long transition sequences [13, 15]. To combat this [13] proposes Deep Contextualized Word Embbedings, a word representation that encodes words with respect to the context of the entire sentence therefore giving some global context to transition-based parser classifiers.

The advantages of deep contextualized word embeddings is that they produce contextualized representations over several layers of abstraction based on the neural network's model's different layers and are pre-trained on corpora much larger than typical treebanks [13].

One popular deep contextualized embedding model is Bidirectional Encoder Representations from Transformers (BERT; [23]). BERT is designed to pretrain deep bidirectional representations from unlabeled text. The pre-trained BERT model can be finetuned with one additional output layer to create state-of-the-art models for a wide range of tasks [23]. [4] applies BERT to transition-based MRP obtaining results similar to those in [13] showing that the global information provided by deep contextualized word embeddings help prevent search errors in greedy parsing.

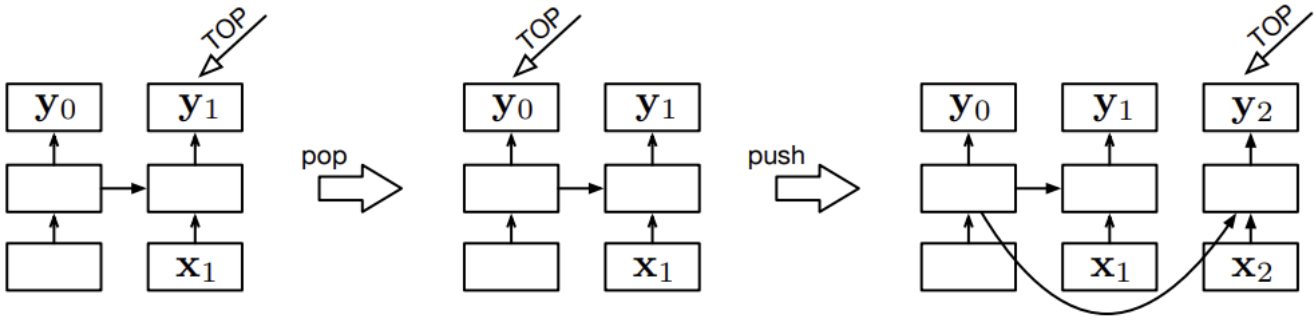

Figure 7: A stack LSTM, TOP represents the stack pointer, the upper row are the outputs of the LSTM, the middle row are memory cells and gates, the bottom row is the input to the LSTM [8]

5.3 State of the Art

The current state of the art transition-based parser for EDS is the HIT-SCIR 2020 system ([7]). HIT-SCIR 2020 is a multi-framework transition-based parser presented at the 2020 Conference for Computational Language Learning (CoNLL; [19]) shared task. The HIT-SCIR 2020 system has different approaches for different frameworks, but for EDS it directly uses the top system from the 2019 CoNLL shared task ([20]), HIT-SCIR 2019 ([4]). The HIT-SCIR 2019 system makes use of stack LSTMs which are explained in a section below (§5.4).

Although HIT-SCIR 2019 is the best transition-based parser for EDS it not the best EDS parser. The Hitachi system ([22]) outperforms HIT-SCIR 2019 using a text-to-graph-notation transducer, leveraging transformers and biaffine attention, the details of which are beyond the scope of this paper.

We now turn our attention to another parser that leverages transformers, [9] proposes a stack transformer architecture (described in §5.5) that achieved state of art results (later improved by [14]) for Abstract Meaning Representation (AMR; [2]), another MRP framework [19].

It can be seen that transformers are achieving state of the art results in many applications [9, 19, 22]. There is therefore a clear gap where transformers have not been applied to transition-based EDS parsing beyond using deep contextualized word representations [4].

5.4 HIT-SCIR 2019 Architecture

The architecture of HIT-SCIR 2019 is shown in Figure 6. HIT-SCIR feeds BERT contextualized word representations into a transition-based parser to construct a general graph as well as additional tagger models to predict various node label [4]. Since HIT-SCIR 2019 is a multi-framework parser, the construction of the specific MRP framework graphs is done as a post processing step [4]. We will focus on the transition-based parser in this section.

The transition-based parser represents the parsing state as a tuple (S, L, B, E, V) where S is a stack holding processed words, L is a list holding words popped out of S that will be pushed back later, B is a buffer holding unprocessed words, E is a set of labeled dependency arcs, and V is a set of nodes [4]. S, L, B and the action history is modelled with stack LSTMs (described in §5.5) based on the work of [8]. The outputs of these stack LSTMs are then passed to a classifier that outputs an action that maximizes the score using the following formula where a is an action and s is the parser state represented by the stack LSTMs:

$$p(a|s) = \frac{\exp\{g_a \cdot \texttt{STACK LSTM}(s) + b_a\}}{\sum_{a'} \exp\{g_{a'} \cdot \texttt{STACK LSTM}(s) + b_{a'}\}}$$

5.5 Stack Architectures

5.5.1 Stack LSTMs. A stack LSTM is shown in Figure 7, it consists of a list of entries containing, the input to the LSTM x_i, a memory cell and gates and the LSTM output y_i. A "stack pointer" is used to represent the entry corresponding to the top of the stack, this entry is called the top. The stack LSTM has 3 operations [8]:

- **Push** adds a new entry at the end of the list that contains the next input, a new memory cell that is calculated with the memory cell provided by the current top, and the output calculated with the new memory cell. The pointer then moves to this new entry and a back-pointer to the previous top is added.
- **Pop** moves the stack pointer to the previous element using the back-pointer.
- **Query** Returns the output vector of the top entry.

5.5.2 Stack Transformers. [9] proposes that instead of using a stack LSTM described above to encode the stack and buffer in transitions systems, a transformer can be used instead. [9] takes advantage of how sequences are encoded as a weighted sum of tokens plus positional embeddings as well as the multi-headed attention mechanism to achieve this. Since all tokens are summed there is no need for a stack pointer, an item that is popped off the stack or buffer can just be masked out of the score calculation. To separately model the stack and buffer two heads of the attention mechanisms are specialized to attend to the stack and buffer respectively [9].

6 CONCLUSION

In this paper we have reviewed some of the state of the art transition-based parsers and EDS parsers. Our findings show that transformer based neural network architectures have shown state of the art

results in parsing. Transformers have already been used to achieve state of the art results in transition-based ARM parsing but the current state of the art transition-based EDS parser is still outperformed by other approaches using transformers. There is a gap in the research as transformer neural networks have not yet been applied to transition-based EDS parsing beyond use in deep contextualised word representations.

REFERENCES

[1] Hongxiao Bai and Hai Zhao. 2019. SJTU at MRP 2019: A Transition-Based Multi-Task Parser for 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*. Association for Computational Linguistics, Hong Kong, 86–94. https://doi.org/10.18653/v1/K19-2008

[2] 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*. Association for Computational Linguistics, Sofia, Bulgaria, 178–186. https://aclanthology.org/W13-2322

[3] 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)*. Association for Computational Linguistics, Vancouver, Canada, 1215–1226. https://doi.org/10.18653/v1/P17-1112

[4] 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*. Association for Computational Linguistics, Hong Kong, 76–85. https://doi.org/10.18653/v1/K19-2007

[5] Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 740–750. https://doi.org/10.3115/v1/D14-1082

[6] Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan Sag. 2005. Minimal Recursion Semantics: An Introduction. *Reseach On Language And Computation* 3 (07 2005), 281–332. https://doi.org/10.1007/s11168-006-6327-9

[7] Longxu Dou, Yunlong Feng, Yuqiu Ji, Wanxiang Che, and Ting Liu. 2020. HIT-SCIR at MRP 2020: Transition-based Parser and Iterative Inference Parser. In *Proceedings of the CoNLL 2020 Shared Task: Cross-Framework Meaning Representation Parsing*. Association for Computational Linguistics, Online, 65–72. https://doi.org/10.18653/v1/2020.conll-shared.6

[8] 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)*. Association for Computational Linguistics, Beijing, China, 334–343. https://doi.org/10.3115/v1/P15-1033

[9] Ramón Fernandez Astudillo, Miguel Ballesteros, Tahira Naseem, Austin Blodgett, and Radu Florian. 2020. Transition-based Parsing with Stack-Transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1001–1007. https://doi.org/10.18653/v1/2020.findings-emnlp.89

[10] Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*. The COLING 2012 Organizing Committee, Mumbai, India, 959–976. https://aclanthology.org/C12-1059

[11] Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (1st ed.). Prentice Hall PTR, USA.

[12] Marco Kuhlmann and Stephan Oepen. 2016. Towards a Catalogue of Linguistic Graph Banks. *Computational Linguistics* 42, 4 (12 2016), 819–827. https://doi.org/10.1162/COLI_a_00268 arXiv:https://direct.mit.edu/coli/article-pdf/42/4/819/1807537/coli_a_00268.pdf

[13] Artur Kulmizev, Miryam de Lhoneux, Johannes Gontrum, Elena Fano, and Joakim Nivre. 2019. Deep Contextualized Word Embeddings in Transition-Based and Graph-Based Dependency Parsing - A Tale of Two Parsers Revisited. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 2755–2768. https://doi.org/10.18653/v1/D19-1277

[14] Young-Suk Lee, Ramón Fernandez Astudillo, Tahira Naseem, Revanth Gangi Reddy, Radu Florian, and Salim Roukos. 2020. Pushing the Limits of AMR Parsing with Self-Learning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 3208–3214. https://doi.org/10.18653/v1/2020.findings-emnlp.288

[15] Ryan McDonald and Joakim Nivre. 2007. Characterizing the Errors of Data-Driven Dependency Parsing Models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Association for Computational Linguistics, Prague, Czech Republic, 122–131. https://aclanthology.org/D07-1013

[16] Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*. Nancy, France, 149–160. https://aclanthology.org/W03-3017

[17] Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics* 34, 4 (2008), 513–553. https://doi.org/10.1162/coli.07-056-R1-07-027

[18] Joakim Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. Association for Computational Linguistics, Suntec, Singapore, 351–359. https://aclanthology.org/P09-1040

[19] Stephan Oepen, Omri Abend, Lasha Abzianidze, Johan Bos, Jan Hajic, Daniel Hershcovich, Bin Li, Tim O'Gorman, Nianwen Xue, and Daniel Zeman. 2020. MRP 2020: The Second Shared Task on Cross-Framework and Cross-Lingual Meaning Representation Parsing. In *Proceedings of the CoNLL 2020 Shared Task: Cross-Framework Meaning Representation Parsing*. Association for Computational Linguistics, Online, 1–22. https://doi.org/10.18653/v1/2020.conll-shared.1

[20] 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*. Association for Computational Linguistics, Hong Kong, 1–27. https://doi.org/10.18653/v1/K19-2001

[21] 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)*. European Language Resources Association (ELRA), Genoa, Italy. http://www.lrec-conf.org/proceedings/lrec2006/pdf/364_pdf.pdf

[22] Hiroaki Ozaki, Gaku Morio, Yuta Koreeda, Terufumi Morishita, and Toshinori Miyoshi. 2020. Hitachi at MRP 2020: Text-to-Graph-Notation Transducer. In *Proceedings of the CoNLL 2020 Shared Task: Cross-Framework Meaning Representation Parsing*. Association for Computational Linguistics, Online, 40–52. https://doi.org/10.18653/v1/2020.conll-shared.4

[23] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 2227–2237. https://doi.org/10.18653/v1/N18-1202

[24] Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-Reduce Dependency DAG Parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*. Coling 2008 Organizing Committee, Manchester, UK, 753–760. https://aclanthology.org/C08-1095

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf