IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 July 2022, accepted 30 August 2022, date of publication 8 September 2022, date of current version 15 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3204997

RESEARCH ARTICLE

Syntactic Factors Associated With Performance of Dependency Parsers Using Stack-Pointer Network and Graph Attention Networks Between English and Korean

YONG-SEOK CHOI, YO-HAN PARK, AND KONG JOO LEE
Department of Radio and Information Communications Engineering, Chungnam National University, Yuseong-gu, Daejeon 34134, Republic of Korea

Corresponding author: Kong Joo Lee (kjoolee@cnu.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant through the Korea Government [Ministry of Science and ICT (MSIT)] under Grant NRF-2019R1F1A1053136, and in part by the ''Regional Innovation Strategy (RIS)'' through the National Research Foundation of Korea (NRF) through the Ministry of Education (MOE) under Grant 2021RIS-004.

ABSTRACT A Stack-Pointer Network (StackPtr) parser is a pointer network with an internal stack on the decoder. Several studies use the StackPtr as the backbone of a dependency parser because it can traverse a parse tree depth-first without backtracking and can handle high-order parsing information easily thanks to the internal stack. The parser can use information from previously derived subtrees stored in the internal stack upon selecting a child node. In this work, we introduce a new StackPtr parser with Graph Attention Networks (GATs) that can encode a previously derived subtree. We evaluated our proposed parser on the Sejong and Everyone's corpora for Korean and on the Penn Treebank and Universal Dependency corpora for English. In addition, we analyzed and compared our proposed parser with other variants of the StackPtr parser, examining the syntactic information that each parser can reference at every decoding step. We found that Korean parse trees tend to have more consecutive immediate single-child nodes than English parse trees. The proposed StackPtr parser with GATs performed best on almost all metrics for Korean because it can utilize more context to analyze these parse trees by grasping Korean syntactic factors than any other variants. However, for English, no particular variant of the StackPtr parser outperforms the others.

INDEX TERMS Dependency parsing, graph attention networks, natural language processing, stack-pointer networks, high-order dependency parsing.

I. INTRODUCTION

Dependency parsing can identify a syntactical relationship between words in a sentence; it is a basic method in natural language processing. The main approaches to dependency parsing can be divided into two categories: graph-based parsing and transition-based parsing.

Graph-based parsing [1], [2], [3] scores every pair of words in a sentence and then generates a fully connected parse tree based on the scores in a greedy style. Scoring a word pair is a key component of a graph-based parser. One of the most common approaches to graph-based parsing is to use a biaffine parsing model [1]. In this model, a bi-directional recurrent neural network converts a word from a sentence into an embedded vector. Then, biaffine attention networks calculate a dependency score for an edge between the parent and child vectors. Connecting edges individually according to the high scoring order can determine the final dependency tree.

Transition-based dependency parsers [4] build a dependency tree incrementally by scanning the words of a sentence individually and making a sequence of decisions. The key to transition-based parsing models is to use as many features as possible in each decision step to avoid leading to a wrong dependency tree.

A recent approach in transition-based dependency parsing is to use Stack-Pointer Networks (StackPtr) [5], which this

The associate editor coordinating the review of this manuscript and approving it for publication was Juan Wang.

work aims to upgrade. A StackPtr parser analyzes a sentence in an incremental, top-down, and depth-first fashion. A dependency parser based on the StackPtr consists of an encoder and a decoder. The encoder encodes every word of an input sentence while the decoder chooses a dependency in each step. This is done by pointing to a child word in the encoder given a parent word in a stack of the decoder. The stack of the decoder maintains the parent information from previously derived subtrees in parsing. Therefore, the parser can easily utilize high-order information in the decoding step by using parent, grandparent, and sibling words as the parser's features.

Thanks to the success of the StackPtr parser, several variants of the StackPtr parse, including Hierarchical StackPtr [6], and the left-to-right version of the StackPtr parser [7] have been proposed.

In this paper, we introduce Graph Attention Networks (GATs) [8] into the StackPtr parser to take advantage of a previously derived subtree structure in a decoding phase. The GATs can encode a subtree maintained in the decoder's internal stack, and then the decoder takes the encoded subtree information as input.

As the GATs increase the number of layers, they can easily encode multi-hop neighboring information from a graph. Therefore, by stacking multiple layers of the GATs, the proposed parser can gradually utilize various higher-order features more easily than other variants of StackPtr parsers.

Moreover, GATs can represent a node by aggregating its neighboring nodes according to their attention values, indicating the importance between nodes. Therefore, the StackPtr parser with the GATs can encode a subtree taking into account the importance of the nodes. However, other variants of StackPtr parsers incorporate high-order information merely by summing node vectors in a stack without considering structure or importance values.

We compared our proposed parser and the variants on evaluation of Korean and English treebanks. The experimental results showed that the proposed parser performs best on almost all metrics for Korean and no variant of the StackPtr parser performs overwhelmingly better than others for English. We conducted a comprehensible error analysis and investigated which factors of the GATs contribute to improving Korean parsing accuracy. Korean parse trees tend to have more consecutive immediate single-child nodes than English parse trees. The GATs allow the StackPtr parser to leverage more contextual information when analyzing these Korean parse trees than when analyzing English parse trees.

The contributions of this paper are as follows:

1) We introduce GATs into the StackPtr parser to encode a previously derived subtree structure.
2) We show that the StackPtr parser with the GATs can achieve state-of-the-art performance for Korean.
3) We explore the syntactic factors that affect the performance of the proposed parser by comparing it with other variants of StackPtr parsers and analyzing the characteristics of Korean syntactic trees.

The rest of the paper is organized as follows. We first explore related studies in Section 2, and we propose a StackPtr parser with the GATs in Section 3. Section 4 and 5 present an experimental environment and its main results. Finally, Section 5 describes the conclusion of this study.

II. RELATED WORKS

There are two mainstream methods in the dependency parsing research domain. One is graph-based parsing and the other is transition-based parsing.

Graph-based parsers normally score all word pairs to determine the dependency relationship between a word pair. The scoring function calculates relatedness based on two vectors corresponding to parent and child nodes in a dependency tree. Therefore, how to represent a node is a key component of graph-based parsers.

[2] used graph neural networks(GNNs) to learn the node representations. GNNs can embed a node by aggregating node information on its neighbors and collecting more neighbors with multiple hops incrementally. To differentiate between a parent and child node, they adopted two multilayer perceptrons to generate a parent node and child node for a node pair, respectively. The scoring function is basically a bilinear function of a parent and child node. Given a complete graph, the scoring function can provide different weights to all possible edges. By starting with a complete graph, different weights on the edges can convert the complete graph into a soft parse tree in each layer.

The biaffine parser proposed in [1] is one of the most popular graph-based dependency parsers. Although the biaffine parser is simple and efficient, it supports only first-order parsing that can decide every dependency by considering only a parent and child pair. [3] proposed an efficient and effective tri-affine operation for scoring second-order subtrees to extend the biaffine parser to a high-order parsing model. In addition, they adopted a second-order TreeCRF for structural learning in dependency parsers. They demonstrated that structural learning and high-order modeling can enhance the state-of-the-art biaffine parser by conducting several experiments using 13 languages.

Transition-based parsers build a dependency tree by applying transition operations incrementally. Their advantage lies in linear time complexity, however, their main weakness is the lack of access to global context information, which leads to error propagation. There are many previous studies to alleviate the error propagation problem in transition-based dependency parsing. One of the successful approaches is a StackPtr parser [5] that adopts a top-down depth-first strategy to perform the syntactic analysis. The StackPtr parser using a pointer network has achieved state-of-the-art performance because the pointer network with an internal stack can capture information of the input sentence and all the paths previously traversed. After the successful implementation of StackPtr in dependency parsing, several subsequent studies have been conducted.

[6] proposed a dependency parser based on hierarchical StackPtr. The parser has the same encoder-decoder architecture as the original StackPtr parser except that each decoding state is conditioned directly on a hierarchical tree structure albeit its sequential processing. Therefore, their parser could explicitly model the parent-child and sibling relationships in the decoding steps.

Another variant of the StackPtr parser is a left-to-right dependency parser using StackPtr, proposed in [7]. While the original StackPtr parser analyzes a sentence in a top-down depth-first manner, the left-to-right StackPtr parser processes an input sentence one by one from left to right. In a decoding phase, the original StackPtr parser points to a child word, however, the decoder of the left-to-right StackPtr parser points to a parent word for each hidden state. This results in a straightforward transition system that can reduce the number of transition operations, without the need of any additional data structures. Instead, the left-to-right StackPtr parser might generate a cycle, therefore, it needs an additional cycle detection step. In spite of the simplicity of the algorithm, surprisingly, the parser achieved the better performance than the original StackPtr parser.

III. PARSER BASED ON STACK-POINTER NETWORKS WITH GRAPH ATTENTION NETWORKS

The StackPtr parser [5] consisting of an encoder and a decoder scans a complete sentence and encodes each word into an encoder hidden state s_i. The StackPtr parser then initializes an internal stack of the decoder with a root symbol "\$" and generates a dependency tree in a top-down fashion by pointing to a hidden state of the encoder one by one, as shown in Fig. 1.

At the top of the internal stack of the decoder is the most recently derived child word of a sub-tree, which becomes a parent word in the next decoding step. In each decoding step, an embedding vector corresponding to the top word of the stack becomes an input of the decoder and attention scores between the embedding and all hidden states of the encoder are calculated. The decoder points out the hidden state with the highest attention score as a child word and then pushes the child word to the top of the stack for the next step. When the decoder points to the same word as the top word, it indicates that all children of the top word have already been derived. In this case, the decoder pops the top element from the stack and parsing continues to the next step.

How to represent an embedding vector corresponding to the top word of the internal stack is the main concern of this paper. For a first-order parsing model, the original StackPtr parser uses the encoder's hidden state corresponding to the parent word. The parser uses a summation of the corresponding hidden states for a second- or third-order model that uses a sibling or grand-parent word and a parent word.

In this study, we adopted the GATs [8] to generate an embedding vector for a top word of the stack. The GATs can generate an embedding of a previously derived subtree centered on a top word. In each decoding step, an embedding for a partially derived subtree becomes an input of the decoder.

A. StackPtr PARSER WITH GATs

In this section, we describe a dependency parser using the StackPtr as a backbone and the BERT [9] pre-trained language model and then introduce the GATs into the StackPtr parser.

The StackPtr parser consists of an encoder and a decoder. The decoder has an internal stack that controls its inputs. The main role of the encoder built on long short-term memory (LSTM) networks [10] is to generate contextualized word embeddings from an input sentence. We used the BERT pre-trained model to obtain basic word embeddings in this work. A unit the BERT model uses is a WordPiece [11], which is different from a word that is an input unit of a syntactic parser.

Given that a word usually consists of several wordpieces, they should be converted into one input embedding for parser's inputs. Let W $= w_1, w_2, \ldots, w_M$ be an input sentence with M words and X $= \{x_1, x_2, \ldots, x_N\}$ be a sequence of N wordpieces of the sentence W split for the BERT [9] model. When Z $= \{z_1, z_2, \ldots, z_N\}$ is the BERT output embedding of input X, we can obtain an embedding of word w_i by summing up embedding z_t corresponding to x_t contained in w_i (1).

$$\hat{z}_i = \sum_{x_t \in w_i} z_t \tag{1}$$

The encoder of the StackPtr parser is a bi-directional LSTM network whose t-th hidden state s_t is obtained as (2). Let S $= \{s_1, s_2, \ldots, s_M\}$ be the hidden state of the encoder.

$$\begin{aligned} \overrightarrow{s}_t &= \text{LSTM}(\hat{z}_t, \overrightarrow{s}_{t-1}), \quad \overleftarrow{s}_t = \text{LSTM}(\hat{z}_t, \overleftarrow{s}_{t+1}), \\ s_t &= [\overrightarrow{s}_t || \overleftarrow{s}_t] \end{aligned} \tag{2}$$

The main role of the decoder built on LSTM networks is to determine a child word in every decoding step in a top-down way. From a parser's pointer of view, a hidden state of the decoder indicates a parent word and a hidden state of the encoder pointed out by the decoder is a child word. The encoder's hidden states S are also used as inputs of a hidden state of the decoder. If the hidden state at decoding step t was h_t and s_i was an encoder's hidden state of a candidate child word, a biaffine attention mechanism was applied to calculate an attention score e_i^t between two hidden states h_t and s_i that correspond to a parent and a child word, respectively.

$$e_i^t = h_t^T W s_i + U^T h_t + V^T s_i + b \tag{3}$$

$$p^t = \text{softmax}(e^t), \tag{4}$$

where W,U,V, and b are learnable parameters, and the attention scores e^t collected for S are transformed to probabilities by the softmax function. The encoder's hidden state with the highest probability is determined as a child word. As we need dependency labels besides the dependency relationship, we adopted an additional biaffine attention network [1] with

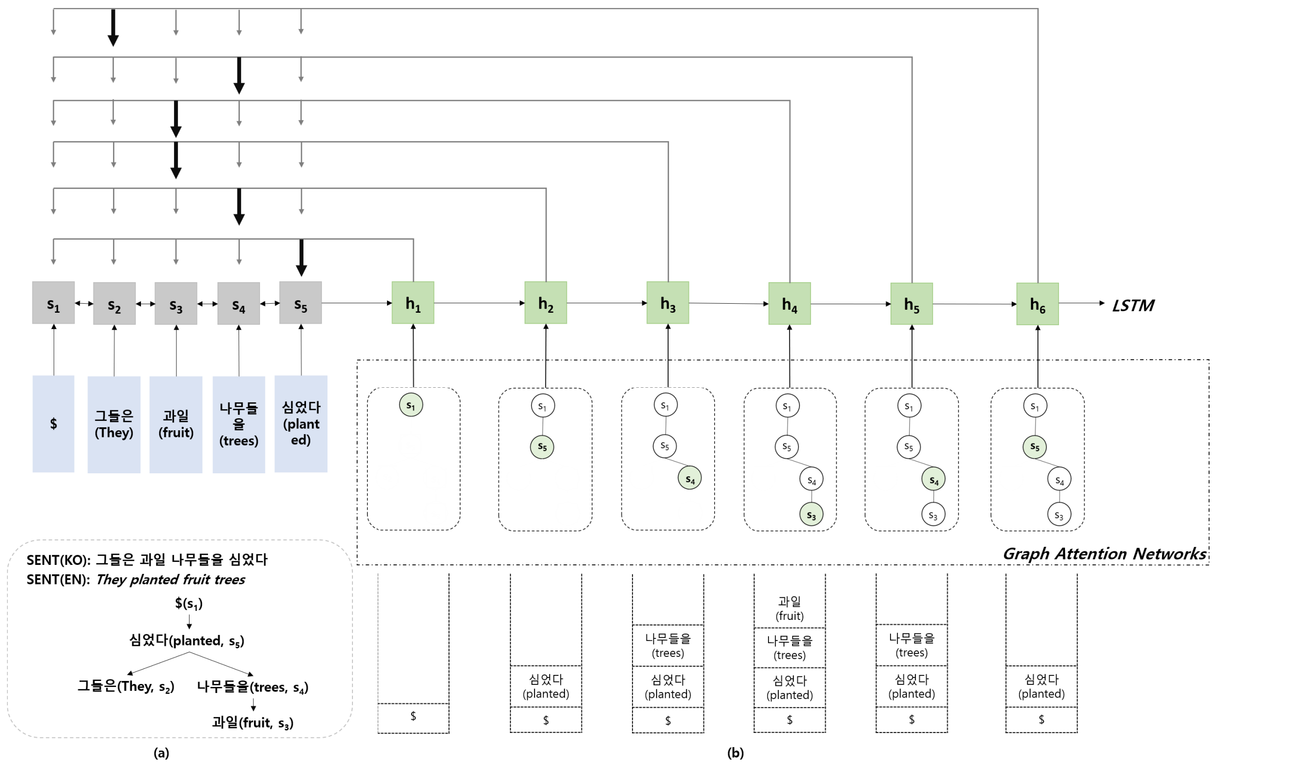


FIGURE 1. Dependency parsing using a StackPtr with the GATs for the sentence "They planted fruit trees". (a) is the correct dependency tree of the input sentence, and (b) is the architecture of the StackPtr parser with GATs. The hidden state h_6 of the decoder has a parent node corresponding to "planted" in the top of the stack and GATs can encode the parent node corresponding to the state s_5 using a previously derived subtree. The hidden state h_6 can point at its child node, which is the state s_2 of the encoder in this case.

the same architecture as that in (3) to generate dependency labels. To utilize higher-order information in dependency parsing, the original StackPtr parser uses a summed vector of hidden states corresponding to a parent, grandparent, and sibling word.

However, using the simple summed vector of the hidden states is not enough to perform high-order parsing. Therefore, in this study, we introduce GATs [8] to a parsing model to encode higher-order syntactic information. The GATs take an encoder's hidden states S and a subtree $T_{(t-1)}$ derived up to $t-1$ step as inputs and produce a node representation d_t stuffed with the subtree's structural information. The new node representation is fed into the LSTM cell in the decoding step t, as shown in (5) and (6). We will describe this in detail in the following section.

$$d_t = \text{GATs}(S, T_{t-1}) \tag{5}$$

$$h_t = \text{LSTM}(d_t, h_{t-1}) \tag{6}$$

B. REPRESENTING A SUBTREE USING THE GATs IN THE StackPtr PARSER

Fig. 2 shows the method to generate a node $d_{(t,5)}$ for the t-step hidden state of the decoder. For brevity, we drop a step subscript t in Fig. 2 and the following explanation. The GATs [8] generate an embedding for node d_5 by aggregating neighboring nodes according to their attention score α. As the

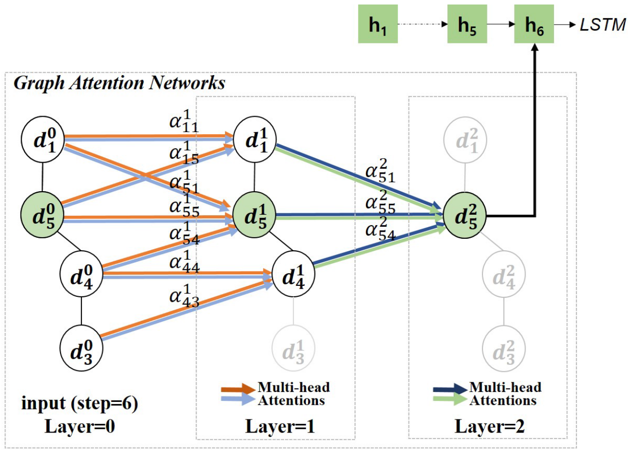


FIGURE 2. Node representation using the GATs for input into the 6th step (hidden state h_6) of the decoder in Fig. 1. A node d_i^0 in the zero-th layer is the hidden state s_i of the encoder.

GATs have two layers in this example, all neighboring nodes within two hops affect d_5, which is notated as d_5^2. Given an undirected graph G, the GATs that are a multilayer network provide us with a node representation contextualized by its neighboring nodes in graph G. As the number of layers in the GATs increases, the number of contextualized nodes also increases. A node representation can be obtained by calculating the attention scores between two adjacent nodes and then aggregating adjacent nodes according to their attention scores.

We will start by describing the GATs with a single attention and then extend it to multi-head attention to stabilize the learning process of attentions.

In (7), g_{ij} is an attention coefficient that indicates the importance of node j's features to node i, where $W \in R^{dim_{out} \times dim_{inp}}$ is a weight matrix applying the same linear transformation to every node and $A \in R^{2*dim_{out}}$ is a shared weight matrix for computing the attention coefficient. We set dim_{inp} to 512 and dim_{out} to 32 in the experiments. In (8), α_{ij} is an attention score normalized by all its neighboring nodes and N_i is a set of neighboring nodes of node i (including node i) in the graph.

$$g_{ij} = \mathrm{f}(A^T[Wd_i||Wd_j]) \tag{7}$$

$$\alpha_{ij} = \mathrm{softmax}(g_{ij}) = \frac{\exp(g_{ij})}{\sum_{k \in N_i} \exp(g_{ik})} \tag{8}$$

In (9), d_i^l represents node d_i in layer l of the GATs. Nodes in the zero-th layer are defined as $d^0 = \{s_1, \ldots, s_M\}$, which denotes the hidden states of the encoder. W^l and α_{ij}^l are a weight matrix W and an attention score α_{ij} corresponding to the layer l, respectively. The node embedding d_i^l is calculated by aggregating the embeddings of the adjacent nodes of node d_i by the attention values. Now, we enhance the model to include H multi-head attentions in the GATs. The embedding d_i^l is concatenated by the outputs of the multi-head attentions, as shown in (10). The embedding d_i^l becomes an t-step input d_t to the LSTM cell of the decoder.

$$d_i^l = \sigma(\sum_{j \in N_i} \alpha_{ij}^l (W^l d_j^{l-1})) \tag{9}$$

$$d_i^l = \|_{h=1}^{H} \sigma(\sum_{j \in N_i} \alpha_{ij}^{h,l} (W^{h,l} d_j^{l-1})) \tag{10}$$

C. TRAINING OBJECTIVES

The training objective function of the proposed parser is the same as that of the original StackPtr parser. The parser predicts an arc first and then predicts a label for the arc. The objective function is to minimize a weighted combination of arc prediction (L_{arc}) and label prediction (L_{label}) as in (13).[1] The loss function of arc prediction is (11), in which x is an input and $p_{<t}$ denotes the preceding paths that have already been generated prior to time t and p_t is a child node representation predicted by the parser at time t. Given a head node representation h_t at time t and a predicted child p_t, a label classifier is optimized using the cross-entropy loss in (12). Both objectives are calculated based on a biaffine attention score between two vectors corresponding to a head word and child word.

$$L_{arc} = -\sum_{t=1}^{T} \log P_\theta(p_t | p_{<t}, x) \tag{11}$$

[1] $\lambda_1 = \lambda_2 = 1$

TABLE 1. Information from the Sejong, Everyone's, Penn Treebank, and UD corpora.

Corpus		Number of sentences	Avg. number of words	Avg. number of morphemes
EVERYONE's NIKL (2020,ver 1.0)	Train	119,500	13.29	30.31
	Test	15,000	13.33	30.50
	Dev	14,830	13.03	29.72
SEJONG (2010)	Train	53,842	11.19	24.89
	Test	5,817	9.93	22.21
PENN Treebank (v3.0)	Train	39,832	23.85	NA
	Test	2,416	23.46	NA
	Dev	1,700	23.59	NA
UD Treebank (v2.3)	Train	12,543	16.31	NA
	Test	2,077	12.08	NA
	Dev	2,002	12.56	NA

$$L_{label} = -\sum_{t=1}^{T} \log P_\theta(r | p_t, h_t) \tag{12}$$

$$L = \lambda_1 L_{arc} + \lambda_2 L_{label} \tag{13}$$

IV. EXPERIMENTS

A. TRAINING CORPORA

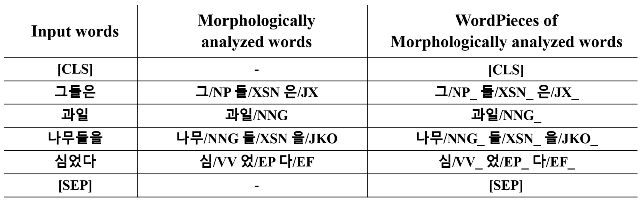
We used two versions of the Korean treebank – the Sejong corpus [12] and the Everyone's corpus [13], [14] – which were published by National Institute of Korean Language (NIKL). Given that the syntactic annotation scheme of the Sejong corpus consists of phrase structures, we converted the structures into dependency structures. Table 1 describes the statistical information on the Everyone's and Sejong corpora.

We evaluated our model on the English Penn Treebank (PTB v3.0) [15], which was converted into the Stanford dependencies format [16]. In addition, we performed experiments on the English language from the Universal Dependency (UD) treebanks.[2] The UD treebanks were built to facilitate multilingual parser development. The UD treebanks provide a universal inventory of categories and guidelines to facilitate consistent annotation of similar constructions across languages while allowing language-specific extensions when necessary. As of November 2019, the UD treebanks version 2 (UD v2.3) [3] is available. In the experiments, we adopted the standard training/dev/test splits and used the universal POS tags provided in the English UD treebank (UD v2.3). Table 1 describes the statistical information on the Penn and UD treebanks.

B. PRE-TRAINED WORD EMBEDDINGS

To initialize word vectors, we used the pre-trained word representation model, BERT [9]. As Korean is an agglutinative language in which suffixes generally take a syntactic role, those suffixes need to be separated from the rest of a word for efficient parsing. Therefore, we performed a morphological analysis of Korean sentences before parsing. Then, we applied the KorBERT morphology version built on morphologically analyzed sentences to obtain the initial word

[2] http://universaldependencies.org
[3] https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-3105



Input words	Morphologically analyzed words	WordPieces of Morphologically analyzed words
[CLS]	-	[CLS]
그들은	그/NP 들/XSN 은/JX	그/NP_ 들/XSN_ 은/JX_
과일	과일/NNG	과일/NNG_
나무들을	나무/NNG 들/XSN 을/JKO	나무/NNG_ 들/XSN_ 을/JKO_
심었다	심/VV 었/EP 다/EF	심/VV_ 었/EP_ 다/EF_
[SEP]	-	[SEP]

FIGURE 3. The wordpieces of the sentence, "They planted fruit trees" for input into KorBERT and the parser. (NNG and NP are parts of speech of general nouns and pronouns, respectively; JX and JKO are auxiliary and objective case particles, respectively; XSN is a noun-derived suffix; EP and EF are past-tense endings and final endings, respectively.)

vectors of a sentence. The morphology version of KorBERT, which uses WordPiece as a unit, contains 30,349 wordpieces. Fig. 3 provides an example of an input to the parser.

We did not use the BERT model for English sentences; instead, we used structed-skipgram word embeddings [17] adopted in the StackPtr parser [5] to compare with previous research.

C. TRAINING

The model parameters and the training hyper-parameters used in the experiments are shown in Table 2. We used the same experimental settings as [5]; therefore, the same experiments are conducted with three repetitions to report the average accuracy for the UD treebank, Sejong, and Everyone's corpora, and with five repetitions for the Penn treebank.

TABLE 2. Model parameters of the StackPtr parser with GATs and hyper-parameters for training setup.

Modules	Parameters	Values
Encoder	Mode	Bi-LSTM
	Number of layers	3
	Hidden dimension	256
Decoder	Mode	Uni-LSTM
	Number of layers	2
	Hidden dimension	256
	Arc dimension	512
	Label dimension	128
GATs	Hidden dimension	512
	Number of attention heads	16
Training (hyper-parameters)	Dropout	0.1
	Batch size	32
	Optimizer	Adam
	Maximum learning rate of encoder	2e-5
	Maximum learning rate of decoder	1e-3
	Beta1, Beta2	0.9, 0.998
	Learning rate scheduler	Exponential LR
	Warm-up steps	40

V. MAIN RESULTS

A. PARSING ACCURACY

Parsing performances were measured using the following metrics: unlabeled attachment score (UAS), labeled attachment score (LAS), unlabeled complete match (UCM), and labeled complete match (LCM), which are the de facto standard to evaluate performance of syntactic parsers. The UCM and LCM indicate how completely a parser can analyze an input sentence.

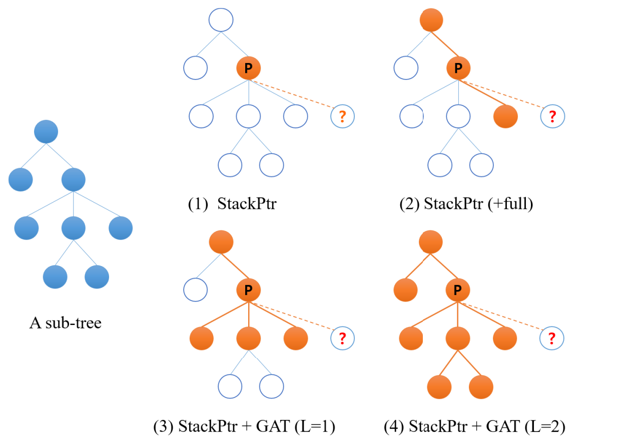


FIGURE 4. Contextual information on the high-order StackPtr parser. Assume the leftmost one is a partially derived sub-tree up to the decoding step *t* and (1)-(4) show the possible high-order parsing in the decoding step $t+1$. A node labeled "P" is a parent word, and a node with a question mark is a candidate of a child word in the decoding step $t+1$. From a parsing perspective, nodes used as features to decide a child node in the step $t+1$ are shown in red.

Table 3 summarizes the performance results of the StackPtr parsers applied to both Korean and English. The experiments are conducted on the Sejong corpus [12] and the Everyone's corpus [13], [14] for Korean and on the Penn Treebank [18] and the UD treebanks for English. We have six variants, (a) – (f) of the StackPtr parsers to compare. The StackPtr parsers (a), (b), and (c) are from [5]. The basic StackPtr parser (a) only utilizes a parent word when deciding its child, (b) utilizes a parent and an additional sibling node, and (c) utilizes a parent, sibling, and grandparent together. The hierarchical StackPtr parser (d) is from [6], and the left-to-right StackPtr (e) is from [7]. The StackPtr with GATs (f) is ours, and "L" is the number of layers of the graph attention networks.

In Korean, the proposed parser, "StackPtr + GATs," mostly achieved higher scores than the other variants of the StackPtr parser according to most metrics. However, in English, we cannot assert that any specific StackPtr parser remarkably outperforms the other StackPtr variants.

B. COMPARISON OF StackPtr PARSERS

Fig. 4 compares of the original StackPtr parser and the proposed parser in terms of the parsing order, which is the number of dependencies referenced in parsing. (1) and (2) are the first- and third-order parsing of the basic StackPtr parser, respectively. (3) and (4) are the higher-order parsing of the StackPtr using GATs with one and two layers, respectively.

$$h_6 = \mathrm{LSTM}(s_5, h_5) \tag{14}$$

$$h_6 = \mathrm{LSTM}(s_1 \oplus s_5 \oplus s_4, h_5) \tag{15}$$

$$h_6 = \mathrm{LSTM}(d_5^2, h_5) \tag{16}$$

$$h_6 = \mathrm{LSTM}(s_5, \mathrm{f}(h_5, h_1, h_2)) \tag{17}$$

TABLE 3. Comparison of parsing performance of the proposed model. For the parsing results on the PennTree bank, the scores of (a), (b) and (c) are from [5] and those of (d) are from [6] and those of (e) are from [7], respectively.

			Parsers					
Lang.	Corpus	Metrics	(a) **StackPtr**	(b) **StackPtr** (+sib)	(c) **StackPtr** (+full)	(d) **StackPtr** + Hierarchical decoder	(e) Left-to-Right **StackPtr**	(f) **StackPtr** + GAT (L=2) (ours)
Korean	Sejong	UAS	94.13	94.31	94.36	94.33	94.26	**94.39**
		LAS	92.46	92.56	92.62	92.65	92.67	**92.71**
		UCM	66.31	66.96	66.99	67.13	66.77	**67.27**
		LCM	**59.86**	59.36	59.12	59.69	59.53	59.58
	Every one's	UAS	93.05	93.46	93.51	93.55	93.60	**93.76**
		LAS	90.71	91.34	91.32	91.39	91.45	**91.63**
		UCM	49.36	49.62	49.83	50.12	50.18	**51.62**
		LCM	38.65	39.85	39.87	40.16	40.35	**41.79**
English	Penn Tree	UAS	95.77	95.85	95.87	**96.09**	96.04	95.71
		LAS	94.12	94.18	94.19	**95.03**	94.43	94.52
	UD (2.3)	UAS	**91.45**	91.36	90.97	91.03	91.31	90.96
		LAS	**89.25**	89.24	89.06	89.07	89.20	88.65

To show the difference between the original StackPtr and hierarchical StackPtr parser [6], (14) – (17) specifically describe the calculation of the hidden state h_6 of the decoder in Fig. 1.

Equation (14) shows the basic original StackPtr parser, in which the decoder's hidden state h_6 is the output of the LSTM cell computed with the previous hidden state h_5 and the parent state s_5 of the encoder.

Equation (15) shows the original StackPtr (+full) version with the sum of the parent s_5, grandparent s_1, and sibling s_4 states. Equation (16) shows the StackPtr parser + GATs (L = 2), in which d_5^2 is illustrated in Fig. 2. The node d_5^2 is the parent state s_5 aggregated with the neighboring nodes (s_1, s_4, and s_3) using the GATs introduced in this study. In the hierarchical StackPtr parser [6], the next hidden state of the decoding step depends not only on the previous hidden state but also the parent's and sibling's decoder states. Equation (17) shows the hierarchical StackPtr parser, in which h_1 is the parent's hidden state and h_2 is the sibling's hidden state, respectively.

C. ERROR ANALYSIS AND DISCUSSION

From the results displayed in Table 3, we intend to determine through our analysis why the StackPtr + GATs parser outperforms the hierarchical StackPtr and the StackPtr (+full) parsers in Korean.

Fig. 5 shows the comparison of characteristics of dependency trees between Korean and English. Surprisingly, 57.64 – 60.45% of all nodes in the Korean dependency trees have immediate adjacent parents in sentences, whereas 37.54 – 43.01% of nodes have adjacent parents in English. In addition, Korean has more only-child nodes (without siblings) than English. About 46% of all nodes are only-child nodes in the Korean dependency trees, while 16.15 – 24.85% are only-child nodes in English. Korean is known to be a head-final language in which a parent follows its children in a sentence. Based on Korean's head-final property, we can draw a typical Korean dependency tree (a) (or (b)) as shown in Fig. 6, whose characteristics correspond to the facts observed in Fig. 5.

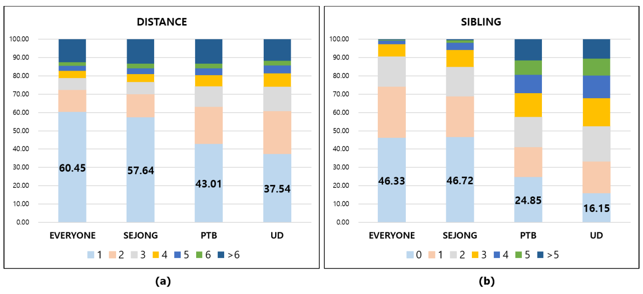


FIGURE 5. Comparison of the characteristics of dependency trees of Korean (Everyone's and Sejong corpora) and English (Penn Treebank and UD corpora). (a) The proportion of the distance between a node and its parent in a sentence; (b) the proportion of the number of sibling nodes in a dependency tree according to the corpus.

In Fig. 6, when node g attaches to node f as a single child, the information that each parser can exploit is quite different, especially when a dependency tree has sequential single-child nodes. The hierarchical StackPtr parser can only refer to the encoder's state s_2 and the decoder's hidden state h_9, as shown in Equation (f). The StackPtr (+full) parser can use the encoder states s_2 and s_3, and the hidden state h_9, as shown in Equation (g). The StackPtr + GATs (L = 2) can refer to a node d_2^2 and the hidden state h_9 as shown in Equation (e). The node d_2^2 is constructed by the GATs using the encoder states s_2, s_3, and s_7. We can reason that the StackPtr + GATs can utilize more information to determine a correct dependency relation than the hierarchical StackPtr and the StackPtr (+full) can when sentences have linear structures that are more common in Korean. Therefore, we cautiously conclude that the StackPtr + GATs can perform better in Korean than other variants of StackPtr parsers.

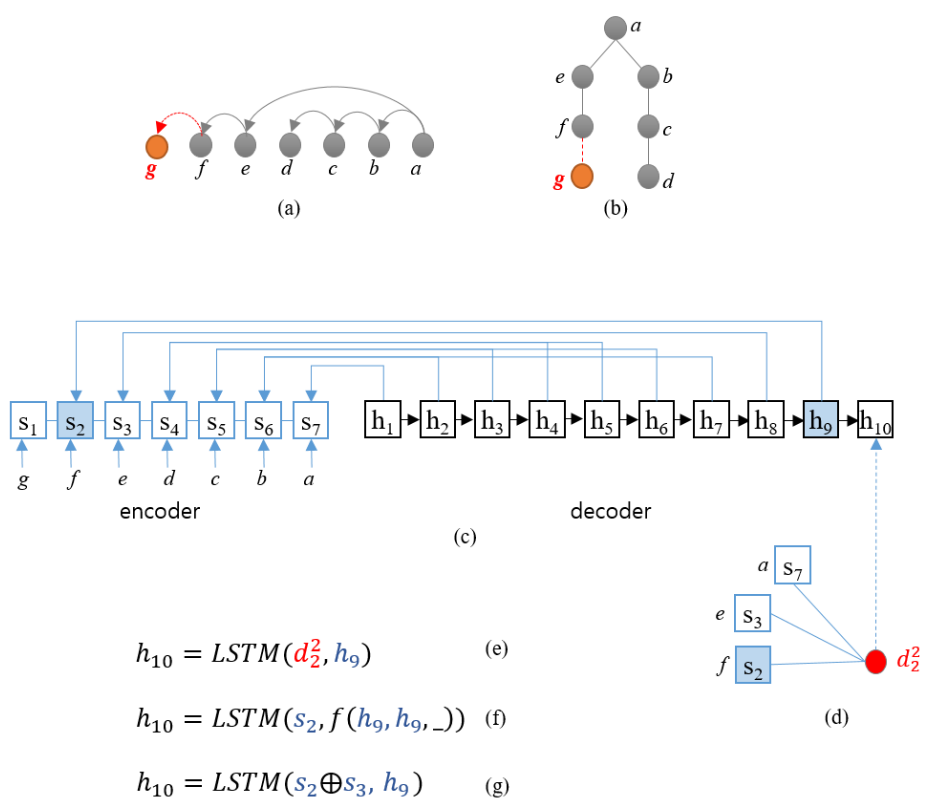

g f e d c b a
(a)
a b e f c g d
(b)
s1 s2 s3 s4 s5 s6 s7
g f e d c b a
encoder
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
decoder
(c)
a s7
e s3
f s2
d_2^2
(d)
$h_{10} = LSTM(d_2^2, h_9)$ (e)
$h_{10} = LSTM(s_2, f(h_9, h_9, \_))$ (f)
$h_{10} = LSTM(s_2 \oplus s_3,\ h_9)$ (g)

FIGURE 6. Illustration of the StackPtr parser's processing of trees (a) and (b), which are the same. Tree (a) is an example of a dependency tree with several immediate adjacent parents and single-child nodes. (c) is the status of an encoder and decoder of a basic StackPtr parser. (d) is the GAT node constructed by the StackPtr + GATs. Equations (e), (f), and (g) are for the hidden state h_{10}, calculated by StackPtr + GATs, Hierarchical StackPtr and the StackPtr(+full).

VI. CONCLUSION

This paper proposed a StackPtr parser with the GATs to encode a sub-tree derived up for every decoding step.

The GATs can support a node representation in a graph, in which each node is calculated using its neighboring nodes multiplied by weights indicating their importance. When the GATs have several layers, more neighboring nodes with multiple hops can be used to represent the node. From a parsing perspective, the StackPtr parser with the GATs can deal with parent, child, and sibling nodes different by considering their importance. It can also incorporate high-order parsing information easily.

Several variants of the StackPtr parser have been proposed, including ours, and their relative performances are compared herein. Through error analysis of the parsing results, we found that Korean dependency trees tend to have consecutive immediate single-child nodes and the StackPtr parser can be used with the GATs to parse these trees. Therefore, the proposed StackPtr parser with GATs achieved the best performance for Korean. However, the original StackPtr and the hierarchical StackPtr parsers outperform the proposed StackPtr parser in English.

Our future study will include how to find syntactic factors, if any, that affect performance of parsers.

REFERENCES

[1] T. Dozat and C. D. Manning, "Deep biaffine attention for neural dependency parsing," 2016, *arXiv:1611.01734*.
[2] T. Ji, Y. Wu, and M. Lan, "Graph-based dependency parsing with graph neural networks," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 2475–2485. [Online]. Available: https://aclanthology.org/P19-1237
[3] Y. Zhang, Z. Li, and M. Zhang, "Efficient second-order TreeCRF for neural dependency parsing," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, Jul. 2020, pp. 3295–3305.
[4] C. Dyer, M. Ballesteros, W. Ling, A. Matthews, and N. A. Smith, "Transition-based dependency parsing with stack long short-term memory," 2015, *arXiv:1505.08075*.
[5] X. Ma, Z. Hu, J. Liu, N. Peng, G. Neubig, and E. Hovy, "Stack-pointer networks for dependency parsing," 2018, *arXiv:1805.01087*.
[6] L. Liu, X. Lin, S. Joty, S. Han, and L. Bing, "Hierarchical pointer net parsing," 2019, *arXiv:1908.11571*.
[7] D. Fernández-González and C. Gómez-Rodríguez, "Left-to-right dependency parsing with pointer networks," 2019, *arXiv:1903.08445*.
[8] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. 6th Int. Conf. Learn. Represent.*, 2018, pp. 1–10.
[9] J. Devlin, M.-W. Chang, K. Lee, and K. N. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, 2018, pp. 4171–4186.
[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[11] Y. Wu, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.

[12] "21st century Sejong project," Nat. Inst. Korean Lang., Seoul, South Korea, Tech. Rep., 2010.

[13] "Everyone's national institute of the Korean language morphological analysis corpus (version 1.0)," Nat. Inst. Korean Lang., Seoul, South Korea, Tech. Rep., 2020. [Online]. Available: https://corpus.korean.go.kr/

[14] "Everyone's national institute of the Korean language parsing corpus (version 1.0)," Nat. Inst. Korean Lang., Seoul, South Korea, Tech. Rep., 2020. [Online]. Available: https://corpus.korean.go.kr/

[15] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: The Penn treebank," *Comput. Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[16] M.-C. de Marneffe and C. D. Manning, "The Stanford typed dependencies representation," in *Proc. Workshop Cross-Framework Cross-Domain Parser Eval.* Manchester, U.K.: Coling Organizing Committee, Aug. 2008, pp. 1–8. [Online]. Available: https://aclanthology.org/W08-1301

[17] W. Ling, C. Dyer, A. W. Black, and I. Trancoso, "Two/too simple adaptations of word2vec for syntax problems," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, 2015, pp. 1299–1304.

[18] D. Chen and C. Manning, "A fast and accurate dependency parser using neural networks," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 740–750. [Online]. Available: https://aclanthology.org/D14-1082

YONG-SEOK CHOI received the B.S. degree in information communications engineering and the M.S. degree in electronics, radio and information communications engineering from Chungnam National University, Daejeon, Republic of Korea, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree with the Department of Electronics, Radio and Information Communications Engineering. His research interests include natural language processing, Korean text processing, syntactic parsing, and machine translation.

YO-HAN PARK received the B.S. degree in radio and information communications engineering from Chungnam National University, Daejeon, Republic of Korea, in 2020, where he is currently pursuing the M.S. and Ph.D. degrees with the Department of Radio and Information Communications Engineering. His research interests include natural language processing, Korean text processing, and machine translation.

KONG JOO LEE received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 1992, and the M.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 1994 and 1998, respectively.

From 1998 to 2003, she was a Researcher at Microsoft, South Korea. Since 2005, she has been a Professor with the Department of Radio and Information Communications Engineering, Chungnam National University, Daejeon. Her research interests include natural language processing, Korean text processing, syntactic parsing, machine translation, and information extraction.