Effective Online Reordering with Arc-Eager Transitions

Ryosuke Kohita    Hiroshi Noji    Yuji Matsumoto
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, Japan
{kohita.ryosuke.kj9, noji, matsu}@is.naist.jp

Abstract

We present a new transition system with word reordering for unrestricted non-projective dependency parsing. Our system is based on decomposed arc-eager rather than arc-standard, which allows more flexible ambiguity resolution between a local projective and non-local crossing attachment. In our experiment on Universal Dependencies 2.0, we find our parser outperforms the ordinary swap-based parser particularly on languages with a large amount of non-projectivity.

1 Introduction

A dependency tree as illustrated in Figure 1 is called a non-projective tree, which contains discontinuous subtrees and is informally remarked with crossing arcs (arcs from idea to talking and from who to to). Comparing to the class of projective trees, which has a weak equivalence to the context-free grammars (Gaifman, 1965), parsing non-projective trees is generally involved. This is particularly the case for transition-based dependency parsing; contrary to the graph-based approaches, in which a simple spanning-tree algorithm is capable of handling them (McDonald et al., 2005), due to the incremental nature, transition-based parsers need some extra mechanisms to find crossing arcs.

There are several attempts to handle crossing arcs in transition-based parsers. Among them online reordering with swap (Nivre, 2009) has a number of appealing properties, of which the most notable is that it inherits the standard architecture of the transition systems using a stack and buffer while covering all types of crossing arcs. This simplicity allows us to incorporate the ideas developed for the standard projective parsers, such as neural network architectures (Chen and Manning, 2014; Dyer et al., 2015), and joint modeling with other phenomena (Hatori et al., 2011; Honnibal and Johnson, 2014), with a minimal effort. Such extensions with swap include a recent non-projective neural parser (Straka et al., 2015) and joint system with POS tagging (Bohnet and Nivre, 2012). Other approaches often employ additional data structures with non-trivial transitions (Covington, 2001; Choi and McCallum, 2013; Pitler and McDonald, 2015), which interfere with the transparency to the standard systems, or cannot handle all crossing arcs (Attardi, 2006).

Despite the popularity of the swap system, to our knowledge there is little work focusing on the swap mechanism, or the transition system itself, apart from the original proposal (Nivre, 2009; Nivre et al., 2009). In other words, we are still unsure whether the current swap mechanism is the best strategy for handling crossing arcs with word reordering.

In this work, we present a dependency parser with a new transition system that employs swap-based reordering but in a different manner from the existing one (Nivre, 2009) built on the arc-standard system. As we discuss (Section 2.2), in Nivre’s transition system, choosing a correct swap transition is sometimes hard due to the parser’s preference to local attachments. The proposed system (Section 3) alleviates this difficulty by allowing a swap transition for a token that is already linked on the stack. As we will see, it can be seen as an extension to the arc-eager system (Nivre, 2003) while we divide each attachment transition into two more primitive operations as in the divided formulation of Gómez-Rodríguez and Nivre (2013). The divided system is more flexible, and by operating swap on this we can deal with the issue of reordering at an appropriate step.

On this transition system we implement a pars-
ing model with the stack LSTMs (Dyer et al., 2015) (Section 4). We extensively examine the utility of new transition system (Section 5) with the recently released Universal Dependencies (UD) 2.0 dataset, which contains more than 60 treebanks with varying degree of non-projectivity, and find that our system is superior to the ordinary swap system particularly for languages with a larger amount of non-projectivity.

2 Background

We first introduce some notations and the concept of transition systems (Section 2.1), and then describe the existing swap-based transition system of Nivre (2009) (Section 2.2).

2.1 Transition System

We focus in this paper on a standard transition system operating on a triple called a configuration $c = (\sigma, \beta, A)$, where $\sigma$ is a stack, $\beta$ is a buffer, and $A$ is a set of labeled arcs. See Nivre (2008) for the other variants and overview. In a configuration $i$-th token in a sentence is denoted by its index $i$ while 0 denotes the special root token. Following the standard notations, by $\sigma|i$ and $j|\beta$, we mean $i$ and $j$ are the top-most tokens of the stack and buffer, respectively. We use $i \xrightarrow{l} j$ or $(i, l, j)$ to denote an arc from $i$ to $j$ with label $l$.

Given a sentence of length $n$, the system begins parsing with the initial configuration $c_0 = ([0], [1, 2, 3, \ldots n], \emptyset)$ where only the root token is on the stack, all inputs are on the buffer, and the set of arcs is empty. Parsing finishes when it reaches a terminal configuration, in which any transitions cannot be performed, and the set of arcs $A$ defines a labeled dependency tree. The system continues to make a transition decision at each step under the current configuration until it reaches a terminal configuration.

2.2 Arc-Standard Swap

Arc-standard swap system (ASS) (Nivre, 2009) is the most popular transition system for non-projective dependency parsing, which can handle arbitrary crossing arcs. ASS rows in Table 1 show the set of transitions, in which LA, RA, and SH are the transitions of the arc-standard system, which can only produce a projective tree by linking two adjacent tokens on the top of the stack. Swap (SW) is the key transition to support non-projectivity, which reorders the top two tokens on the stack by moving the second top token back to the buffer. Reducing a reordered token by LA or RA means we create subtrees that are non-adjacent with each other, i.e., crossing arcs.

One potential issue in ASS is its tendency to prefer local attachments due to the mechanism of LA and RA, which at the same time reduce the dependent token. This is problematic in that because crossing arcs often involve a longer dependency arc, if two tokens on the stack are locally likely to be connected, choosing correct SW rather than LA and RA is quite difficult.

To see an example, let us consider the configuration in Figure 2 $(c = (\sigma|\text{who talks}, to|\beta, A))$, which occurs when parsing the sentence in Figure 1. The correct action here is SW to create a crossing arc $\text{talking}_8 \xrightarrow{\text{obl}} \text{who}_5$. However, at this point LA is a more likely transition since $\text{talking}_8 \xrightarrow{\text{obl}} \text{who}_5$ is a typical arc in a relative clause. The problem is that since LA reduces talking, we will miss the arc $\text{who}_5 \xrightarrow{\text{case}} \text{to}_9$ if we choose LA rather than SW.

3 New System: Stay-Eager Swap

Now we describe our proposed transition system, which we call Stay-Eager Swap (SES). We first present the transitions and its advantage (Section 3.1), and then discuss oracles (Section 3.2) and some improvements (Section 3.3).

3.1 Transition System

SES rows in Table 1 show the set of transitions of the new system. To understand the mechanism, we first note that without SW, this system looks very much similar to the arc-eager transition system (Nivre, 2003). The main difference from the original one is in the attaching transitions, which we call STAY-LEFT (SL) and STAY-RIGHT (SR) and do not reduce a dependent token, but just establish an arc between two tokens on the stack top and buffer top. Specifically, this system is identical to the divided arc-eager transition system with the primitive operations in Gómez-Rodríguez and Nivre (2013); in the arc-eager system, LEFT-ARC first builds an arc and then reduces the top of the stack, i.e., it is a combination of $\text{SL} \rightarrow \text{RD}$ (reduce) in our system, while RIGHT-ARC builds an arc and shift the top token of the buffer, i.e., it can be seen as $\text{SR} \rightarrow \text{SH}$.

1 To make this system without SW to fully mimic the original arc-eager system, we need a constraint to prohibit
I have no idea who you are talking to.

We do not reach the detailed description and formal definition of projective order for SES, but they are the one of important

Figure 1: A non-projective sentence.

Table 1: The set of transitions for arc-standard swap (ASS) and stay-eager swap (SES).

3.2 Static and Non-static Oracles

An oracle for a transition system is a function from a configuration to the action that leads to a given dependency tree. Before discussing in details, we first note that our system suffers from the spurious ambiguity as in the arc-eager system (Goldberg and Nivre, 2012), which means an oracle for some configurations is not unique.

Table 2 shows a specific oracle, which checks for each action in descending order whether the current configuration satisfies the condition, and select the first found one.  

Figure 2 shows how our system can swap after resolving local projective attachments.

2The priority of attaching transitions over SW is also helpful to avoid unnecessary SW transitions for nested non-projective structures. For example in Figure 1, if to_b has a child node x_{i.5} at the left to who, and SW is preferred than SL and SR, it causes an additional crossing between talking_{i} → who_{b} and to_{b} → x_{i.5}. This is not the case for our oracle because talking_{i} → who_{b} has been already attached and isCross(i, j) ignores such attached arcs.

3The function isCross(i, j) for SW could be defined like ASS style which reorder by projective order (Nivre, 2009). However their projective orders are different: for example in Figure 1, while to_b comes up at sixth word for ASS, who_b goes down at eighth word for SES. In this paper, we could not reach the detailed description and formal definition of projective order for SES, but they are the one of important
Table 2: A static oracle for our transition system. \( A_g \) is the set of gold arcs. \( \cdot \) means an arbitrary value.

| Transition | Configuration | Condition |
|------------|---------------|-----------|
| STAY-LEFT; | \((\sigma[i, j] \beta, A)\) | \((j, l, i) \in A_g\) |
| STAY-RIGHT; | \((\sigma[i, j] \beta, A)\) | \((i, l, j) \in A_g\) |
| SWAP | \((\sigma[i, j] \beta, A)\) | isCross(i, j) |
| REDUCE | \((\sigma[i, j] \beta, A)\) | \((\cdot, i, j) \in A \land \forall h, j, i) \in A_g \rightarrow (i, h, j) \in A\) |
| SHIFT | \((\sigma[i, j] \beta, A)\) | \(\forall i, l, j \in A_g \land j < i \rightarrow (j, l, i) \in A\) |

\footnote{isCross(i, j) returns true if \( i \) and \( j \) are two endpoints of two crossing arcs yet unattached. Formally: \( (\exists b. b \in \beta \land ((b, \cdot, i) \in A_g \land (b, \cdot, i) \notin \beta \lor (i, \cdot, b) \in A_g \land (i, \cdot, b) \notin \beta) \land (\exists s.s \in \sigma \land ((s, \cdot, j) \in A_g \land (s, \cdot, j) \notin \sigma \lor (j, \cdot, s) \in A_g \land (j, \cdot, s) \notin \sigma)).\)

Table 3: Static oracle transitions by our system for the sentence in Figure 3.

| \( l \) | Transition | Stack | Buffer | Added Arc |
|-------|------------|-------|--------|-----------|
| 0     | SH         | [0]   | [1, 2, 3, 4, 5] |          |
| 1     | SH         | [0, 1] | [2, 3, 4, 5]    |          |
| 2     | SW         | [0]   | [2, 1, 3, 4, 5] |          |
| 3     | SR         | [0]   | [2, 1, 3, 4, 5] | (0, root, 2) |
| 4     | SH         | [0, 2] | [1, 3, 4, 5]    |          |
| 5     | SH         | [0, 2, 1] | [3, 4, 5] | (3, obj, 1) |
| 6     | SL         | [0, 2, 1] | [3, 4, 5] |          |
| 7     | RD         | [0, 2] | [3, 4, 5] | (2, xcomp, 3) |
| 8     | SR         | [0, 2] | [3, 4, 5] |          |
| 9     | SH         | [0, 2, 3] | [4, 5] |          |
| 10    | RD         | [0, 2] | [4, 5] |          |
| 11    | SR         | [0, 2] | [4, 5] | (2, nmod, 4) |
| 12    | SH         | [0, 2, 4] | [5] |          |
| 13    | SR         | [0, 2, 4] | [5] | (4, punct, 5) |
| 14    | RD         | [0, 2] | [5] |          |
| 15    | SH         | [0, 2, 5] | [5] |          |
| 16    | RD         | [0, 2] | [5] |          |
| 17    | RD         | [0]   | [5] |          |

This is a static oracle in that it is a deterministic function given a configuration. Table 3 shows the oracle transitions by this for the sentence in Figure 3. In addition to the static oracle, we also try a partially non-static oracle, which occasionally prefers SH over RD when both are applicable. Specifically, for this oracle when both conditions for SH and RD are satisfied we choose SH with some probability. This allows the parser to learn the transitions that postpone RD when possible, but stochastically, which we found effective in many languages in practice. This is a partially non-static oracle since it does not postpone the other transitions such as SL and SR. Designing such oracle could also be possible; for example, in Figure 2, we can also build the gold tree by SW followed by SH and SR. We leave such fully non-static oracle as well as the dynamic oracle (Goldberg and Nivre, 2012) as a future work.

3.3 Auxiliary transitions

Our system employs the following two additional transitions (AUX in Table 1), which can be applied in restricted conditions.

**UNSHIFT** The arc-eager system is not guaranteed to output a single rooted tree, i.e., it may keep unconnected tree fragments in the stack while the buffer is empty (Nivre, 2008). Then, a parser becomes stuck because no actions are permissible before reaching a terminal configuration, and our system suffers from the same issue. To escape from this, we employ the same hack as Nivre and Fernández-González (2014) and add a special transition UN, which pushes back the stack top node to the empty buffer. We only apply UN at decoding. It is deterministically chosen in the configuration \( c = ([n], [], A \notin ([\cdot, \cdot], n)) \), so does not have any associated score.

**RIGHT-ROOT** This is our new transition to improve the root attachment accuracy for arc-eager. The arc-eager system attaches the sentence root to the special ROOT eagerly immediately after it collects all its left dependents, but this decision is sometimes hard for some types of garden-path. The purpose of RR is to postpone the decision of this root token until the terminal configuration, as in the arc-standard system.

To be concrete, during training, we allow the parser to select SH with some probability when the gold transition is SR_root. This eventually leads to a terminal configuration where the sentence root token not attached to ROOT remains on the stack. RR is used only for this configuration, also during decoding. Note that unlike UN, the parser ex-
explores this transition during training and learns the parameters associated with it. We hope by this the parser becomes capable of postponing the decision on the root token during decoding when it seems ambiguous locally. We use this transition only with the non-static oracle.

4 Parser Model

A model of a transition-based parser calculates the score of each transition at the current configuration. Our model is basically the stack-LSTM parser (Dyer et al., 2015)\footnote{Please refer Ballesteros et al. (2015), Ballesteros et al. (2016), Ballesteros et al. (2017) for the latest version which is sophisticated in some architectures (e.g. character information, dynamic oracle).}, which we slightly customize from the original architecture (Section 4.2). In this work we focus our attention on the incremental setting, in which the model is not able to access the full tokens in the buffer. With regard for transition-based parsers this is practically a more important scenario where the graph-based parser is not applicable.

4.1 Stack-LSTM parser

We first briefly describe the original model in Dyer et al. (2015) designed for the arc-standard system. For a configuration $c_t$ at time $t$, the parser maintains the three vector representations, $s_t$, $b_t$, and $a_t$, that encode the stack, buffer, and action history. Each of them is modeled with a stack LSTM, an LSTM that supports push and pop operations by keeping the representations of intermediate time steps. The stack LSTM for $s_t$ is left-to-right while that for $b_t$ is right-to-left. $a_t$ encodes the entire action history from the initial action to the last action. Using these representations we encode the configuration into a single vector:

$$p_t = \text{ReLU}(W[s_t; b_t; a_t] + e_p),$$

where $W$ is the parameters. Here and the followings $e_z$ denotes a bias vector.

Using this the probability for each valid transition $z_t$ is obtained with restricted softmax:

$$p(z_t | p_t) = \frac{\exp(g_z^T p_t + q_z_t)}{\sum_{z' \in \mathcal{A}(c_t)} \exp(g_{z'}^T p_t + q_{z'})},$$

where $g_z$ are the parameters and $q_z$ is the bias term for action $z$. The set $\mathcal{A}(c_t)$ returns the set of valid transitions on $c_t$. After each transition we update $s_{t+1}$, $b_{t+1}$, and $a_{t+1}$ accordingly following the new configuration.

For the buffer, each element of the LSTM is a token representation, which Dyer et al. (2015) obtains from the word and POS embeddings. $b_t$ is then the last output of the LSTM.

For the stack, each element of the LSTM is a compositional representation of a subtree, or a token if it is just a shifted one. The subtree representation of the stack element is updated in a recursive manner. In Dyer et al. (2015) when LA or RA builds an arc $h \xrightarrow{l} d$, the representation $h$ for the subtree rooted at $h$ is updated by:

$$h' = \tanh(U[h; d; l] + e_h),$$

in which $d$ is the representation for the subtree rooted at $d$, $l$ is the label embeddings. $s_t$ is then updated by popping the top two elements of the stack LSTM, $h$ and $d$, and pushing $h'$.

4.2 Modifications

We modify the above basic architecture in the following three ways.

Configuration encoding This is a restriction that we impose on the model. While the original model exploits the entire sentence for the buffer representation $b_t$ using the LSTM, this violates our assumption of incrementality, the main advantage of the transition-based parsers. We thus avoid to use $b_t$ and instead use the representations of top three nodes on the buffer: $b_{1t}$, $b_{2t}$, and $b_{3t}$. We also use the representations of the top three nodes (subtrees) on the stack, $s_{1t}$, $s_{2t}$, and $s_{3t}$, which we found effective. The new encoding is:

$$o = W[s_t; a_t; s_{1t}; s_{2t}; s_{3t}; b_{1t}; b_{2t}; b_{3t}] + e_p,$n

$$p_t = \text{ReLU}(o).$$

Token representation Many languages in UD are annotated with XPOS, fine language specific tags, as well as FEATS, the morphological features. We utilize the embeddings of these features, initialized randomly. We also add character embeddings, which we obtain from character-level bidirectional LSTMs. Our token representation is:

$$x = \text{ReLU}(V[w; t; tx; f; w_{ch}] + e_x),$$

where $w$, $t$, $tx$, and $f$ are word, POS, XPOS, and FEAT embeddings, respectively. $w_{ch}$ is the output of linear mapping from the concatenation of the last hidden states of the forward and backward character-level LSTMs.
**Composition** This is the only modification needed to obtain the stack representation in our stay-eager transition system. The subtree representation in Dyer et al. (2015) is fully compositional in that \( h \) in Eq. 1 encodes the entire subtree with the recursive network. This is possible essentially because of the bottom-up nature of the arc-standard system. Unfortunately the same encoding is not straightforward in our system due to its arc-eager property, in which the right arcs are constructed top-down (Nivre, 2004). In this work, we give up the full compositionality of the original model, and simply mimic Eq. 1 with the following equation:

\[
c' = \tanh(U[h; d; l; c] + e_h). \tag{2}
\]

We update the node representation of both of the stack top and the buffer top. This means that apart from the original model we also update the dependent node with composition. In the equation, \( c' \) is the updated representation of the head or the dependent after SR or SL, which is originally \( c \). For example, after SL, since the stack top becomes dependent, its representation (\( d \)) is updated to \( \tanh(U[h; d; r; d] + e_h) \). Note that without \( c \) in Eq. 2, the representations of two updated nodes are identical. The role of \( c \) is thus to distinguish the two updates for \( h \) and \( d \).

## 5 Experiment

**Data** We use the 63 treebanks in 45 languages in Universal Dependencies 2.0 (Nivre et al., 2017), which are distributed with the training data in the recent shared task in CoNLL 2017 (Zeman et al., 2017). Following the shared task, we focus on real world parsing and assume the raw input text. For all preprocessing (sentence segmentation, tokenization, and tagging), we use UDpipe 1.1 (Straka et al., 2015). We report the official F1 LAS used in the shared task.

**Baseline** To make a comparison between transition systems fair, we implement the arc-standard lazy swap (ASS) parser (Nivre et al., 2009) with almost the same settings as our stay-eager swap (SES) parser including our network architecture (Section 4.2). We also report the scores of UDpipe 1.1, the baseline system in the shared task, although the results may not be directly comparable as they tune several settings including the oracle and learning rate etc. for each language.

**Settings** Our network sizes are: 100 dimensional word embeddings and LSTMs, 50 dimensional POS, XPOS, and FEATS embeddings, and 20 dimensional action and label embeddings, and 32 dimensional character embeddings and bi-LSTMs. We do not use any pre-trained embeddings. We use Adam (Kingma and Ba, 2014) for the optimizer, and set the learning decay of 0.08 and the dropout ratio in LSTMs of 0.33.

In addition to the greedy search, we also try beam search for learning and decoding (beam size is 8). Note that due to swap, each transition sequence may have a different number of actions. We alleviate this inconsistency by ranking with the average scores (Honnibal and Johnson, 2014). For non-static oracles, we set both probabilities to postpone RD and SR\(_{root}\) to 0.33, which works well for the development set.

**Results** The main results are shown in the left columns of Table 4. Comparing to ASS, our non-static SES achieves the higher LAS on average, regardless of search method. In more detail, it is on about half treebanks (27 for greedy and 28 for beam search) that the score improvements from ASS are more than 0.5 points. Also the static SES is not stronger, suggesting that non-static exploration during training is important for our system.

Focusing on the results on only non-projective sentences (right columns), the score improvements get larger: the average LAS difference between non-static SES and SAS is 1.54 points with greedy search, and 1.18 points with beam search.

To further inspect the parser behaviors on non-projective and projective sentences, we next compare the average LAS on a subset of treebanks, which we divide into four groups according to the shared task. For small treebanks without the development set, we randomly divide the training data at 1:9 ratio for development and training. This baseline is competitive to or stronger than the original implementation of Dyer et al. (2015), which also implements arc-standard swap (https://github.com/clab/lstm-parser). Example UAS on development sets (with gold tags) are: Arabic: 80.83 (Dyer et al.) vs. 82.01 (ours); English: 86.71 (Dyer et al.) vs. 85.28 (ours); and German: 82.87 (Dyers et al.) vs. 84.45 (ours). Both employ greedy search. Note that our system does not use the buffer LSTM.

For learning, we find the following heuristics inspired by max-violation (Huang et al., 2012) works well. Our training is basically local with cross-entropy while for each sentence we calculate the max violation point by beam search and use only the prefix until that point. Although this is simpler than global structured learning (Andor et al., 2016), it provides some improvements with much faster training time.
## Table 4: LAS of all sentences and of only non-projective sentences ordered by the ratio of non-projective sentences in the test data. The scores in brackets are the results with beam search.

| Language | Non-proj ratio | All sentences | Only non-projective sentences |
|----------|----------------|---------------|-------------------------------|
| gpc      | 64.60%         | 49.18 (51.28) | 50.10 (53.85)                 |
| la       | 47.50%         | 37.38 (37.82) | 38.31 (38.49)                 |
| gpc_proiel| 37.92%         | 60.67 (63.74) | 60.92 (64.69)                 |
| la_nbach | 35.87%         | 72.29 (75.20) | 71.98 (75.01)                 |
| eu       | 31.80%         | 66.61 (67.67) | 65.87 (69.86)                 |
| en_lines | 29.54%         | 70.37 (72.16) | 71.71 (79.52)                 |
| la_proiel| 28.30%         | 50.56 (54.38) | 53.71 (54.72)                 |
| nl_lavysmall | 28.13%       | 76.48 (76.25) | 77.35 (77.78)                 |
| pt       | 23.69%         | 77.12 (78.01) | 74.89 (75.49)                 |
| de       | 23.23%         | 65.11 (66.94) | 63.35 (66.64)                 |
| nl       | 22.92%         | 64.56 (65.69) | 62.87 (66.86)                 |
| got      | 22.67%         | 57.04 (57.40) | 55.46 (57.71)                 |
| hu       | 21.60%         | 61.64 (62.72) | 60.61 (64.44)                 |
| it       | 19.59%         | 57.50 (60.61) | 57.66 (62.67)                 |
| ur       | 18.88%         | 75.91 (76.70) | 75.34 (76.46)                 |
| sv_lines | 18.38%         | 71.48 (72.77) | 71.18 (75.28)                 |
| et       | 16.87%         | 50.06 (52.60) | 53.36 (55.57)                 |
| da       | 16.46%         | 70.23 (72.03) | 69.63 (70.24)                 |
| sl       | 15.99%         | 78.58 (79.85) | 78.72 (79.49)                 |
| cs_clit  | 15.76%         | 68.78 (68.66) | 69.97 (70.06)                 |
| cs_cac   | 12.74%         | 79.92 (81.56) | 80.10 (81.20)                 |
| sk       | 12.53%         | 85.29 (85.82) | 84.22 (85.56)                 |
| sl_yi    | 12.18%         | 41.45 (42.62) | 42.40 (44.15)                 |
| tr       | 12.10%         | 52.67 (51.56) | 52.00 (52.54)                 |
| cs       | 11.98%         | 77.82 (80.43) | 78.13 (79.62)                 |
| el       | 11.62%         | 76.89 (78.31) | 76.77 (77.72)                 |
| ar       | 11.62%         | 63.79 (64.80) | 64.97 (65.04)                 |
| kk       | 11.56%         | 21.83 (21.75) | 18.43 (19.59)                 |
| fr       | 11.54%         | 77.20 (78.76) | 77.80 (79.76)                 |
| ca       | 10.94%         | 82.94 (83.82) | 83.40 (84.33)                 |
| ga       | 10.13%         | 58.58 (60.20) | 60.50 (61.30)                 |
| es       | 10.09%         | 70.48 (80.42) | 70.64 (80.81)                 |
| no       | 9.65%          | 61.43 (64.67) | 61.34 (62.03)                 |
| sv       | 9.61%          | 73.70 (74.24) | 72.11 (78.20)                 |
| ro       | 9.48%          | 71.08 (73.84) | 72.49 (73.33)                 |
| ru       | 9.21%          | 83.89 (85.53) | 84.26 (84.95)                 |
| no       | 9.20%          | 78.69 (79.52) | 78.26 (78.78)                 |
| hr       | 9.17%          | 73.74 (76.08) | 73.47 (75.63)                 |
| fr       | 8.77%          | 72.72 (78.69) | 72.09 (77.61)                 |
| uk       | 7.95%          | 57.85 (58.67) | 57.98 (59.47)                 |
| no_boomaa| 7.63%          | 79.81 (80.69) | 79.57 (80.37)                 |
| fa       | 7.17%          | 74.83 (77.09) | 74.61 (77.42)                 |
| fi       | 7.02%          | 70.95 (71.03) | 70.01 (72.03)                 |
| lv       | 6.64%          | 55.54 (57.10) | 54.50 (56.76)                 |
| fi       | 6.64%          | 75.17 (72.37) | 70.45 (72.42)                 |
| id       | 5.75%          | 71.48 (73.78) | 71.17 (74.16)                 |
| it      | 4.96%          | 83.06 (84.68) | 83.61 (84.30)                 |
| pl_br    | 4.90%          | 83.59 (84.39) | 83.38 (83.92)                 |
| ug       | 4.78%          | 31.54 (34.02) | 30.93 (30.34)                 |
| sk       | 4.34%          | 70.13 (70.76) | 70.60 (70.99)                 |
| fr_partut| 4.25%          | 75.86 (76.47) | 76.81 (77.10)                 |
| en_partut| 4.00%          | 70.01 (72.61) | 72.43 (73.44)                 |
| en       | 3.71%          | 72.53 (73.56) | 72.51 (73.81)                 |
| vi       | 3.28%          | 36.37 (36.37) | 36.11 (37.00)                 |
| bg       | 2.87%          | 81.24 (82.02) | 79.92 (81.32)                 |
| sv       | 2.30%          | 74.09 (74.67) | 73.65 (74.29)                 |
| he       | 1.83%          | 55.02 (56.58) | 54.91 (56.21)                 |
| zh       | 1.40%          | 53.55 (55.23) | 51.65 (53.94)                 |
| pl       | 0.27%          | 76.80 (78.02) | 77.09 (79.08)                 |
| gl       | 0.00%          | 76.16 (77.43) | 74.72 (77.48)                 |
| ja       | 0.00%          | 71.47 (72.38) | 63.91 (68.59)                 |

### Average

| Language | Non-proj ratio | All sentences | Only non-projective sentences |
|----------|----------------|---------------|-------------------------------|
| Average  | 13.76%         | 76.59 (81.93) | 76.35 (80.88)                 | 86.05 (82.36) |
the ratio of non-projective sentences (Table 5). When evaluating on all sentences (top rows), we can see the larger improvements by the non-static SES in HIGH, MID, and LOW groups (having non-negligible non-projectivity), which confirms the above results. Interestingly, for projective sentences only (mid rows) the scores of SES do not degrade comparing to ASS, or rather improves in all cases. This suggests our transition system also helps to recover the projective arcs.

6 Discussion

While the ordinary reordering-based transition system is built on the arc-standard system, we choose arc-eager as our basic architecture. One reason for this is that decomposing the arc-standard system is more involved than arc-eager; Gómez-Rodríguez and Nivre (2013) observe the RIGHT-ARC in arc-standard would be divided into four transitions including a nontrivial UNSHIFT operation. Otherwise, we need two different reduce operations for each direction, which complicates the system and learning.

Though we have seen the empirical advantage, in terms of the reordering strategy our approach may not be optimal. Consider the sentence in Figure 4, which we borrow from Nivre et al. (2009). Our system needs more swap transitions than the Nivre et al.’s swap-lazy system for this sentence.

In Nivre et al.’s system, swapping Who$_1$ and did$_2$ occurs after you$_3$ is reduced as a dependent of did$_2$. In our system, due to the right top-down nature of arc-eager, we need to build ROOT$_0$ → did$_2$ before did$_2$ → you$_3$. This means we also need an additional swap between Who$_1$ and you$_3$.

Past work shows that a smaller number of swap transitions improves accuracies (Björkelund and Nivre, 2015), and thus it is an important future work to revise our system to minimize the necessary swap transitions. Another direction might be to incorporate our idea to postpone swap transitions into the arc-standard system, possibly with the divided system as we did for arc-eager.

In our system each word is once attached, shifted, and reduced, so the total number of transitions is 3$n$ plus the number of swap transitions. This is greater than Nivre et al.’s system, though we expect this additional cost is not substantial comparing to the other techniques, e.g., beam

Table 5: The average LAS on all, only projective, and only non-projective sentences on the grouped treebanks according to the ratio of non-projective sentences in the test set. The scores in brackets are the results with beam search.

| Sentence | Non-proj ratio | ASS    | SES    | UDpipe |
|----------|---------------|--------|--------|--------|
|          | static        | non-static |        |        |
| All      | HIGH          | 62.07 (63.79) | 62.21 (64.34) | 62.96 (65.09) | 66.42 |
|          | MID           | 67.27 (68.30) | 67.39 (68.35) | 67.64 (69.05) | 69.93 |
|          | LOW           | 73.23 (74.85) | 73.25 (74.50) | 73.73 (75.11) | 75.19 |
|          | VERYLOW       | 67.49 (68.61) | 67.09 (68.18) | 67.65 (68.80) | 69.68 |
| Projective | HIGH         | 66.38 (67.91) | 66.13 (67.92) | 66.85 (68.80) | 70.25 |
|          | MID           | 68.35 (69.27) | 68.32 (69.14) | 68.59 (69.90) | 70.81 |
|          | LOW           | 74.17 (75.73) | 74.09 (75.31) | 74.57 (75.87) | 76.06 |
|          | VERYLOW       | 67.91 (68.96) | 67.46 (68.53) | 68.01 (69.14) | 70.07 |
| Non-projective | HIGH      | 57.19 (59.20) | 58.02 (60.61) | 58.60 (60.98) | 62.17 |
|          | MID           | 62.04 (63.27) | 62.41 (63.99) | 63.02 (64.50) | 65.22 |
|          | LOW           | 66.21 (68.35) | 67.28 (68.63) | 67.66 (69.47) | 68.75 |
|          | VERYLOW       | 59.74 (61.39) | 60.77 (62.40) | 62.56 (62.13) | 62.52 |

1 HIGH (20%–) : gr, la, grc, proiel, la_itth, eu, en_lines, la_proiel, nl_lassysmall, pt, de, gl_treeegal, nl, got, and hu.
2 MID (10%–20%) : cu, ur sv_lines, et, da, sl, cs_cltt, cs_cac, hi, sl_sst, tr, cs, el, ar, kk, fr, ca, ga, and es.
3 LOW (5%–10%) : ro, es_ancora, ko, ru, ru_syntaxrus, no_nynorsk, hr, fr_sequoia, uk, no_bokmaal, fa, fi_itth, lv, fi, and id.
4 VERYLOW (0%–5%) : it, pt_br, ug, sk, fr_rutu, en_partut, en, vi, bg, sv, be, zh, and pl.
5 Gl and ja are excluded when calculating the scores of “non-projective only” (bottom rows), as these treebanks only contain projective sentences in their test data.

Figure 4: Another non-projective sentence.
search with larger beam sizes.

7 Conclusion

We have shown for incremental non-projective parsing, explicitly separating the attachment and reduce transitions alleviates the difficulty of local decisions, and leads to higher parsing accuracies for both projective and crossing arcs. Non-projectivity is prevalent in multilingual parsing beyond the popular languages in the current NLP such as English and Chinese. Also incremental parsing is essential for many online applications, in particular the speech-oriented systems. In this paper, we proposed an alternative to the popular approach for incremental non-projective parsing. There are much rooms for improvements, and this is our first step of reconsidering the optimal mechanism to handle crossing arcs incrementally.

Acknowledgements

This work was supported by JST CREST Grant Number JPMJCR1513, Japan.

References

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, Berlin, Germany, pages 2442–2452. http://www.aclweb.org/anthology/P16-1231.

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X). Association for Computational Linguistics, New York City, pages 166–170. http://www.aclweb.org/anthology/W/W06/W06-2922.

Miguel Ballesteros, Chris Dyer, Yoav Goldberg, and Noah A Smith. 2017. Greedy transition-based dependency parsing with stack lstms. Computational Linguistics.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with lstms. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Lisbon, Portugal, pages 349–359. http://aclweb.org/anthology/D15-1041.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A. Smith. 2016. Training with exploration improves a greedy stack lstm parser. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Austin, Texas, pages 2005–2010. https://aclweb.org/anthology/D16-1211.

Anders Björkellund and Joakim Nivre. 2015. Nondeterministic oracles for unrestricted non-projective transition-based dependency parsing. In Proceedings of the 14th International Conference on Parsing Technologies. Association for Computational Linguistics, Bilbao, Spain, pages 76–86. http://www.aclweb.org/anthology/W15-2210.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. Association for Computational Linguistics, Jeju Island, Korea, pages 1455–1465. http://www.aclweb.org/anthology/D12-1133.

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, pages 740–750. http://www.aclweb.org/anthology/D14-1082.

Jinho Choi and Andrew McCallum. 2013. Transition-based dependency parsing with selective branching. In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, Sofia, Bulgaria, pages 1052–1062. http://www.aclweb.org/anthology/P13-1104.

Michael A. Covington. 2001. A fundamental algorithm for dependency parsing. In Proceedings of the 39th Annual ACM Southeast Conference. pages 95–102.

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, pages 334–343. http://www.aclweb.org/anthology/P15-1033.

Haim Gaifman. 1965. Dependency systems and phrase-structure systems. Information and Control 8(3):304 – 337. https://doi.org/http://dx.doi.org/10.1016/S0019-9958(65)90232-9.

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, pages 959–976. http://www.aclweb.org/anthology/C12-1059.

Carlos Gómez-Rodríguez and Joakim Nivre. 2013. Divisible transition systems and multiplanar dependency parsing. Computational Linguistics 39(4):799–845. http://www.aclweb.org/anthology/J/J13/J13-4002.pdf.

Jun Hatori, Takuya Matsuzaki, Yusuke Miyao, and Jun’ichi Tsujii. 2011. Incremental joint pos tagging and dependency parsing in chinese. In Proceedings of 5th International Joint Conference on Natural Language Processing. Asian Federation of Natural Language Processing, Chiang Mai, Thailand, pages 1216–1224. http://www.aclweb.org/anthology/I11-1136.

Matthew Honnibal and Mark Johnson. 2014. Joint incremental disfluency detection and dependency parsing. Transactions of the Association for Computational Linguistics 2:131–142. https://transacl.org/ojs/index.php/tacl/article/view/234.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, Montréal, Canada, pages 142–151. http://www.aclweb.org/anthology/N12-1015.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Vancouver, British Columbia, Canada, pages 523–530. http://www.aclweb.org/anthology/H/H05/H05-1066.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In Proceedings of the 8th International Workshop on Parsing Technologies (IWPT). pages 149–160.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In Frank Keller, Stephen Clark, Matthew Crocker, and Mark Steedman, editors, Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together. Association for Computational Linguistics, Barcelona, Spain, pages 50–57.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. Computational Linguistics 34(4):513–553. http://www.aclweb.org/anthology/J/J08/J08-4003.pdf.

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, pages 351–359. http://www.aclweb.org/anthology/P/P09/P09-1040.

Joakim Nivre, Željko Agić, Lars Ahrenberg, Lene Antonsen, Maria Jesus Aranzabe, Masayuki Asahara, Luma Ateyah, Mohammed Attia, Aitziber Atutxa, Elena Badnueva, Miguel Ballesteros, Esha Banerjee, Sebastian Bank, John Bauer, Kepa Bengoetxea, Riyaz Ahmad Bhat, Eckhard Bick, Cristina Bosco, Gosse Bouma, Sam Bowman, Aljoscha Burciardt, Marie Candito, Gauthier Caron, Gülsen Cebirolu Eryiit, Giuseppe G. A. Celano, Savas Cetin, Fabricio Chalub, Jinho Choi, Yongseok Cho, Silvie Cinková, Car Çöltekin, Miriam Connor, Marie-Catherine de Marneffe, Valeria de Paiva, Arantza Diaz de Ibarraza, Kaja Dobrovolsic, Timothy Dozat, Kira Droganova, Marhaba Eli, Ali...
