Mining Rules for Rewriting States in a Transition-based Dependency Parser for English

Akihiro Inokuchi and Ayumu Yamaoka
The Institute of Scientific and Industrial Research, Osaka University
8-1 Mihogaoka, Ibaraki, Osaka 567-0047, JAPAN
{inokuchi, yamaoka}@ar.sanken.osaka-u.ac.jp

ABSTRACT
Recently, methods for mining graph sequences have attracted considerable interest in data-mining research. A graph sequence is a data structure used to represent changing networks. The aim of graph sequence mining is to enumerate common changing patterns appearing more frequently than a given threshold in graph sequences. Dependency analysis is recognized as a basic process in natural language processing. In transition-based parsers for dependency analysis, a transition sequence can be represented by a graph sequence, where each graph, vertex, and edge corresponds to a state, word, and dependency, respectively. In this paper, we propose a method for mining rules to rewrite states reaching incorrect final states to those reaching correct final states, from transition sequences of a dependency parser using a beam search. The proposed method is evaluated using an English corpus, and we demonstrate the design of effective feature templates based on knowledge obtained from the mined rules.

KEYWORDS: Dependency Parsing, Graph Sequence, Frequent Pattern Mining.

Proceedings of COLING 2012: Technical Papers, pages 1275–1290,
COLING 2012, Mumbai, December 2012.

1 Introduction

Data mining is the process of mining useful knowledge from large datasets. Recently, methods for mining graph sequences (dynamic graphs (Borgwardt et al., 2006) or evolving graphs (Berlingerio et al., 2009)) have attracted considerable interest from researchers in the field of data mining (Inokuchi and Washio, 2008). For example, human networks can be represented as a graph, where each vertex and edge corresponds, respectively, to a human and a relationship in the network. If a human joins or leaves the network, the numbers of vertices and edges in the graph increase or decrease, respectively. A graph sequence is one of the data structures used to represent a changing network. Figure 1(a) shows a graph sequence consisting of four steps, five vertices, and various edges between the vertices. The aim of graph sequence mining is to enumerate subgraph subsequence patterns, an example of which is shown in Fig. 1(b), appearing more frequently than a given threshold in graph sequences. Since the development of methods for mining graph sequences, these methods have been applied, for example, to social networks in Web services (Berlingerio et al., 2009), article-citation networks (Ahmed and Karypis, 2011), and e-mail networks (Borgwardt et al., 2006).

Dependency parsing is considered a basic process in natural language processing (NLP), and a number of studies have been reported (Kudo and Matsumoto, 2002; Nivre, 2008). One reason for the increasing popularity of this research area is the fact that dependency-based syntactic representations seem to be useful in many applications of language technology (Kubler et al., 2009), such as machine translation (Culotta and Sorensen, 2004) and information extraction (Ding and Palmer, 2004). Broadly speaking, dependency parsers can be categorized as transition-based, graph-based, and grammar-based dependency parsers. Transition-based dependency parsers, which are data-driven methods, transit between states in a deterministic way using local state information. If the parser adds an incorrect dependency between words once, it never reaches the correct final state. To reduce such incorrect decisions, the parser can keep track of multiple candidate outputs using the beam search principle, thus avoiding making decisions too early (Zhang and Clark, 2008).

In a transition-based parser, a transition sequence can be represented by a graph sequence, where each graph, vertex, and edge, corresponds to a state, word, and dependency, respectively. By mining characteristic patterns from transition sequences for sentences analyzed incorrectly by a parser, it is possible to design new parsers and generate effective feature templates in the machine learner of the parser to avoid incorrect dependency structures. In this paper, we demonstrate the application of graph sequence mining to dependency parsing in NLP.

We propose a method for mining rewriting rules from transition sequences of an arc-eager dependency parser integrated with the beam search principle for English sentences. The mined rewriting rules can shed light on why incorrect dependency structures are returned by this type of parser. We also present effective feature templates designed according to knowledge obtained from the mined rules, and show the improvement of the parser's attachment score, which is a measure of the percentage of words with the correct heads. To mine such rules, the rules should be human-readable, since they are to be used as inspiration for the engineering

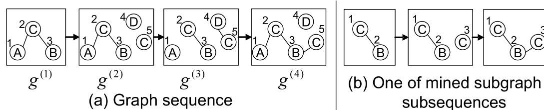


Figure 1: Examples of a graph sequence and a mined frequent pattern

Parse($x = \langle w_0, w_1, \cdots, w_n \rangle$)
1) $c \leftarrow c_s(x)$
2) while $c \notin C_F$
3) $c \leftarrow [o(c)](c)$
4) return c

Figure 2: Dependency parser based on a transition system

Transitions	
Left	$(\sigma\|i, j\|\beta, A) \Rightarrow (\sigma, j\|\beta, A \cup \{(j,i)\})$
Right	$(\sigma\|i, j\|\beta, A) \Rightarrow (\sigma\|i\|j, \beta, A \cup \{(i,j)\})$
Reduce	$(\sigma\|i, \beta, A) \Rightarrow (\sigma, \beta, A)$
Shift	$(\sigma, j\|\beta, A) \Rightarrow (\sigma\|j, \beta, A\})$
Preconditions	
Left	$i \neq 0 \wedge \nexists k$ s.t. $(k,i) \in A$
Right	$\nexists k$ s.t. $(k,j) \in A$
Reduce	$\exists k$ s.t. $(k,i) \in A$

Figure 3: Transitions for an arc-eager parser

of new feature templates of the parser.

2 Transition-based Dependency Parsing

In this paper, we focus on dependency analysis using an "arc-eager parser" (Nivre, 2008), which is a parser based on a transition system, for "English sentences". However, the principle of the method proposed in this paper can basically be applied to any parser based on a transition system for sentences in any language (Inokuchi et al., 2012).

The aim of dependency parsing of a sentence is to output its dependency graph.

Definition 1 *The dependency graph for a sentence* $x = \langle w_0, \cdots, w_n \rangle$ *is represented as a graph* $g = (V, E)$*, where* $V = \{0, \cdots, n\}$ *and* $E \subset V \times V$*.* ■

Definition 2 *A dependency graph* (V, E) *is well-formed, if the following conditions are satisfied:*

- $\nexists x \in V$ *such that* $(x, 0) \in E$ *(root condition),*
- $\nexists x \in V$ *such that* $(x, y) \in E \wedge x \neq x'$*, when* $(x', y) \in E$ *(single-head condition), and*
- $\nexists\{(v_0, v_1), (v_1, v_2), \cdots, (v_{l-1}, v_l)\} \subseteq E$ *such that* $v_0 = v_l$ *(acyclicity condition).* ■

A dependency graph satisfying these conditions is a forest. In addition, if a dependency graph is connected, the graph is a tree.

We define a transition-based dependency parser with input $x = \langle w_0, w_1, \cdots, w_n \rangle$ and output $g = (V, E)$.

Definition 3 *A transition-based parser consists of* $S = (C, T, c_s, C_F)$*, where*

- $C = \{(\sigma, \beta, A)\}$ *is a set of states, with* σ*,* β*, and* A *a stack, a buffer, and a set of edges, respectively,*
- T *is a set of transitions, with* $t \in T$ *a partial function such that* $t : C \rightarrow C$*,*
- c_s *is an initial function satisfying* $c_s(x) = ([0], [1, 2, \cdots, n], \emptyset)$*, and*
- $C_F \subseteq C$ *is a set of final states* $\{c \in C \mid c = (\sigma, [], A)\}$*.* ■

A transition sequence for $x = \langle w_0, w_1, \cdots, w_n \rangle$ on $S = (C, T, c_s, C_F)$ is represented as $C_{1,m} = \langle c^{(1)}, \cdots, c^{(m)} \rangle$, satisfying (1) $c^{(1)} = c_s(x)$, (2) $c^{(m)} \in C_F$, and (3) $\exists t \in T$ for $c^{(i)}$ $(1 \leq i < m)$, $c^{(i+1)} = t(c^{(i)})$. We denote β and A for state c as β_c and A_c, respectively.

Figure 2 gives the algorithm for a transition-based dependency parser, where o denotes an oracle for selecting $t = [o(c)]$ to transit to the next state in a deterministic way. In particular,

the arc-eager parser, which is a transition-based parser, selects either Left, Right, Reduce, or Shift to analyze sentences, as shown in Fig. 3, where the operator | is taken to be left-associative for the stack and right-associative for the buffer. If o selects Left or Right, then edge (j,i) or (i,j) is added to A to transit from c to $t(c)$, respectively. If o selects Reduce, then i is popped from the stack to transit from c to $t(c)$. Otherwise, j is popped from the buffer and j is pushed onto the stack to transit from c to $t(c)$. Since o is a function that determines the transition from Left, Right, Reduce, and Shift, it is implemented using a multi-class classifier for feature vectors characterizing the state c (Kubler et al., 2009).

Although we defined each state using a stack and buffer in a similar way to that reported in most of the literature, we now redefine it using a graph to link dependency parsing to graph sequence mining.

Definition 4 *A state* $c = (\sigma, j|\beta, A)$ *s.t.* $\sigma = [s_{|\sigma|}, s_{|\sigma|-1}, \cdots, s_1]$ *is represented as a graph* $c_g = (N, A, N')$*, where* $N = \{0, 1, \cdots, j\}$ *is a set of vertices,* A *is a set of edges, and* $N' = \{s_{|\sigma|}, s_{|\sigma|-1}, \cdots, s_1, j\} \subseteq N$.

The graph $c_g = (N, A, N')$ is a forest of ordered trees, where $N' \subseteq N$ are vertices that are not reduced on the rightmost path of each tree in the forest. If o selects Left or Right, then edge (j,i) or (i,j), where i and j $(i < j)$ are the largest vertices in N', is added to transit from c to $t(c)$, respectively. If o selects Shift, the smallest vertex that does not exist in N is added to c_g to transit from c to $t(c)$. Otherwise, the second largest vertex in N' is removed from N' to transit from c to $t(c)$. Since an arc-eager dependency parser is incremental (Kubler et al., 2009), the numbers of vertices and edges in c_g increase monotonically.

Example 1 *Figure 4 shows the transition sequence from the initial state to the final state for the sentence* $\langle \$, I, saw, him, . \rangle$*, where* $w_0 = \$$ *is a special root vertex. In the sequence, Shift, Left, Right, Right, Reduce, and Right are selected in order by* o*. In this figure, shaded vertices in each state belong to* N'.

Figure 5 shows the search space T for sentence $\langle w_0, w_1, w_2, w_3 \rangle$. The words in each state and all states whose final states are not trees are omitted owing to lack of space. The search space T for the algorithm given in Figs. 2 and 3 is depicted as a single-rooted directed acyclic graph, where states C on $S = (C, T, c_s, C_F)$ are nodes, initial state $c^{(1)}$ is the root node, final states $C_F \subseteq C$ are leaves, and transitions between the states are branches. As shown in Fig. 5, there is only one or a few transition sequences from the initial state to each leaf. Therefore, if an incorrect dependency is added between words once, the parser never reaches the correct final state. To reduce the possibility of such an incorrect decision, the parser can keep track of multiple candidate outputs using the beam search principle and avoid making decisions too early (Zhang and Clark, 2008). Nevertheless, an arc-eager parser incorporating the beam search principle sometimes reaches an incorrect final state.

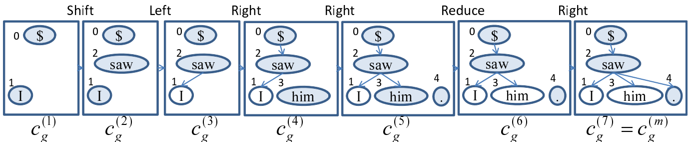


Figure 4: Transition sequence for the sentence $\langle \$, I, saw, him, . \rangle$

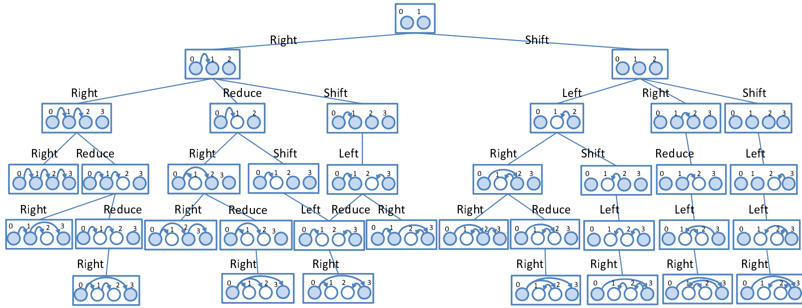
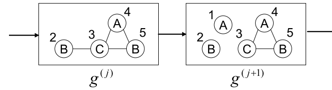


Figure 5: Search space for sentence $\langle w_0, w_1, w_2, w_3 \rangle$



$g^{(j)}$ $g^{(j+1)}$

Figure 6: Change between two successive graphs

In this paper, we propose a method for mining rewriting rules from transition sequences of an arc-eager dependency parser incorporating the beam search principle for English sentences. The rewriting rules to be mined are human-readable rules for rewriting states reaching incorrect final states to those reaching the correct final states. The rewriting rules correspond to bypasses between states in the search space shown in Fig. 5. To describe the proposed method, we first discuss GTRACE for mining graph sequences corresponding to transition sequences in the next section.

3 Graph Sequence Mining

Figure 1(a) shows an example of a graph sequence. Graph $g^{(j)}$ is the j-th labeled graph in the sequence. The problem we address in this section is how to mine patterns that appear more frequently than a given threshold from a set of graph sequences. We proposed using transformation rules to represent graph sequences compactly under the assumption that "change is gradual" (Inokuchi and Washio, 2008). In other words, only a small part of the structure changes, while the other part remains unchanged between successive graphs $g^{(j)}$ and $g^{(j+1)}$ in a graph sequence. For example, the change between successive graphs $g^{(j)}$ and $g^{(j+1)}$ in the graph sequence shown in Fig. 6 is represented as an ordered list of two transformation rules $\langle vi^{(j)}_{[1,A]}, ed^{(j)}_{[(2,3),\bullet]} \rangle$. This list denotes that a vertex with ID 1 and label A is inserted (vi), and then the edge between the vertices with IDs 2 and 3 is deleted (ed). By assuming that the change in each graph is gradual, we can represent a graph sequence compactly, even if the graph in the graph sequence has many vertices and edges. We also proposed a method, called GTRACE (Graph TRAnsformation sequenCE mining), for mining all frequent patterns from ordered lists of transformation rules. A transition sequence in the dependency parser is represented as a graph sequence. In addition, since any change between two successive graphs in the graph sequence comprises at most two changes, the assumption holds.

A labeled graph g is represented as $g = (V, E, L, l)$, where $V = \{1, \cdots, n\}$ is a set of vertices, $E \subseteq V \times V$ is a set of edges, and L is a set of labels such that $l : V \cup E \rightarrow L$. In addition, a graph sequence is an ordered list of labeled graphs and is represented as $d = \langle g^{(1)}, \cdots, g^{(z)} \rangle$.

Table 1: TRs used to represent a graph sequence

Vertex Insertion $vi^{(j,k)}_{[u,l]}$	Insert vertex u with label l into $g^{(j,k)}$ to transform to $g^{(j,k+1)}$.
Vertex Deletion $vd^{(j,k)}_{[u,\bullet]}$	Delete an isolated vertex u in $g^{(j,k)}$ to transform to $g^{(j,k+1)}$.
Vertex Relabeling $vr^{(j,k)}_{[u,l]}$	Relabel the label of vertex u in $g^{(j,k)}$ as l to transform to $g^{(j,k+1)}$.
Edge Insertion $ei^{(j,k)}_{[(u_1,u_2),l]}$	Insert an edge with label l between vertices u_1 and u_2 in $g^{(j,k)}$ to transform to $g^{(j,k+1)}$.
Edge Deletion $ed^{(j,k)}_{[(u_1,u_2),\bullet]}$	Delete an edge between vertices u_1 and u_2 in $g^{(j,k)}$ to transform to $g^{(j,k+1)}$.
Edge Relabeling $er^{(j,k)}_{[(u_1,u_2),l]}$	Relabel a label of an edge between vertices u_1 and u_2 in $g^{(j,k)}$ as l to transform to $g^{(j,k+1)}$.

To represent a graph sequence compactly, we focus on the differences between two successive graphs $g^{(j)}$ and $g^{(j+1)}$ in the sequence.

Definition 5 *The differences between graphs $g^{(j)}$ and $g^{(j+1)}$ in d are interpolated by a virtual sequence $d^{(j)} = \langle g^{(j,1)}, \cdots, g^{(j,m_j)} \rangle$, where $g^{(j,1)} = g^{(j)}$ and $g^{(j,m_j)} = g^{(j+1)}$. The graph sequence d is represented by $d = \langle d^{(1)}, \cdots, d^{(z-1)} \rangle$.* ■

The order of graphs $g^{(j)}$ represents the order of the graphs in an observed sequence. On the other hand, the order of graphs $g^{(j,k)}$ is the order of graphs in the artificial graph sequences, and there can be various artificial graph sequences between graphs $g^{(j)}$ and $g^{(j+1)}$. We limit the artificial graph sequences to be compact and unambiguous by taking the one with the shortest length in terms of the graph edit distance to reduce both the computational and spatial costs.

Definition 6 *Let the transformation of a graph by either insertion, deletion, or relabeling of a vertex or an edge be a unit, and let each unit have edit distance 1. A graph sequence $d^{(j)} = \langle g^{(j,1)}, \cdots, g^{(j,m_j)} \rangle$ is defined as an artificial graph sequence in which the edit distance between any two successive graphs is 1 and the edit distance between any two graphs is the minimum.* ■

Transformations are represented in this paper by the following "transformation rule (TR)".

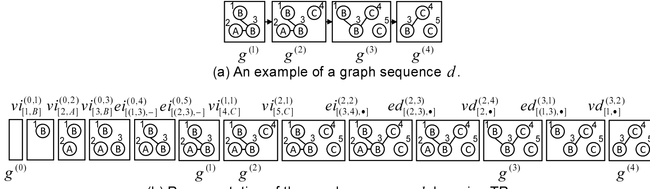
Definition 7 *A TR transforming $g^{(j,k)}$ to $g^{(j,k+1)}$ is represented by $tr^{(j,k)}_{[o_{jk},l_{jk}]}$, where*

- *tr is the transformation type that is either insertion, deletion, or relabeling of a vertex or edge,*
- *o_{jk} is the vertex or edge to which the transformation is applied, and*
- *$l_{jk} \in L$ is a label to be assigned to the vertex or edge in the transformation.* ■

For the sake of simplicity, we simplify $tr^{(j,k)}_{[o_{jk},l_{jk}]}$ to $tr^{(j,k)}_{[o,l]}$ using the six TRs listed in Table 1. In summary, we define a transformation sequence as follows.

Definition 8 *A graph sequence $d^{(j)} = \langle g^{(j,1)}, \cdots, g^{(j,m_j)} \rangle$ is represented by $seq(d^{(j)}) = \langle tr^{(j,1)}_{[o,l]}, \cdots, tr^{(j,m_j-1)}_{[o,l]} \rangle$. Moreover, a graph sequence $d = \langle g^{(1)}, \cdots, g^{(z)} \rangle$ is represented by a transformation sequence $seq(d) = \langle seq(d^{(0)}), \cdots, seq(d^{(z-1)}) \rangle$.* ■

The notation for transformation sequences is far more compact than the original graph-based representation since only differences between two successive graphs in d are kept in the sequence. In addition, any graph sequence can be represented by the six TRs in Table 1.



(a) An example of a graph sequence d.

(b) Representation of the graph sequence d by using TRs.

Figure 7: Graph sequence and its TRs

Example 2 *The graph sequence d in Fig. 7(a) can be represented by a sequence of insertions and deletions of vertices and edges, as shown in Fig. 7(b). The transformation sequence is represented as* $\langle vi^{(0,1)}_{[1,B]} vi^{(0,2)}_{[2,A]} vi^{(0,3)}_{[3,B]} ei^{(0,4)}_{[(1,3),-]} ei^{(0,5)}_{[(2,3),-]} vi^{(1,1)}_{[4,C]} vi^{(2,1)}_{[5,C]} ei^{(2,2)}_{[(3,4),-]} ed^{(2,3)}_{[(2,3),\bullet]} vd^{(2,4)}_{[2,\bullet]} ed^{(3,1)}_{[(1,3),\bullet]} vd^{(3,2)}_{[1,\bullet]} \rangle$, *where "—" denotes an edge label.*

When transformation sequence s'_d is a subsequence of transformation sequence s_d, denoted as $s'_d \sqsubseteq s_d$, there is a mapping ϕ from vertex IDs in s'_d to those in s_d. We omit a detailed definition thereof owing to lack of space (see (Inokuchi and Washio, 2008) for the details). Given a set of graph sequences $DB = \{d \mid d = \langle g^{(1)}, \cdots, g^{(z)} \rangle\}$, we define a support $sup(s_p)$ of transformation sequence s_p as $sup(s_p) = |\{d \mid d \in DB, s_p \sqsubseteq seq(d)\}|/|DB|$. We call a transformation sequence whose support is no less than the minimum support sup', a frequent transformation subsequence (FTS). Given a set of graph sequences, GTRACE enumerates a set of all FTSs from the set according to the anti-monotonic property of the support. GTRACE-RS (Inokuchi et al., 2012) which is an extended version of GTRACE first mines FTSs each of which consists of a TR. It then mines FTSs by recursively adding one TR to the mined FTS.

4 Mining Rules for Rewriting States

As mentioned in Section 2, if the parser shown in Fig. 2 adds an incorrect dependency between words once, it never reaches the correct final state. Even if the parser keeps track of multiple candidate outputs using the beam search principle, sometimes all the multiple candidates reach incorrect final states. In this study, we set out to discover rules for rewriting states reaching incorrect final states to those reaching the correct final states from a corpus $D = \{(x, g)\}$ consisting of sentences $x = \langle w_0, w_1, \cdots, w_n \rangle$ and their dependency graphs g. The rewriting rules state that "if state c_g contains graph p as a subgraph, the state is transformed to another state using a certain TR". The rewriting rules contain knowledge about why the dependency parser outputs the incorrect final states and what should be done to fix these incorrect states. Since p and c_g can also be represented as TRs, each rewriting rule is represented as a sequences of TRs.

To mine these rewriting rules, it needs to be determined how to generate the input graph sequences for GTRACE from transition sequences of the dependency analysis. For the sake of simplicity, we assume that the beam width of the parser is 1. Let $C_{1,m}(x) = \langle c_g^{(1)}, c_g^{(2)}, \cdots, c_g^{(m)} \rangle$ be a transition sequence from the initial state, $c_g^{(1)}$, to the correct final state, $c_g^{(m)} = g$, for sentence $(x, g) \in D$. In addition, let $C'_{1,m'}(x) = \langle c_g^{(1)}, \cdots, c_g^{(k-1)}, c'^{(k)}_g, c'^{(k+1)}_g, \cdots, c'^{(m')}_g \rangle$ be another transition sequence for x, where the parser selects the incorrect transition between $c_g^{(k-1)}$ and $c'^{(k)}_g$ for some $k > 1$. One way of generating a graph sequence from transition sequence $C'_{1,m'}(x)$ is to append the correct final state g to the transition sequence after the parser outputs the incorrect final state; i.e., $d_A = \langle c_g^{(1)}, \cdots, c_g^{(k-1)}, c'^{(k)}_g, c'^{(k+1)}_g, \cdots, c'^{(m')}_g, g \rangle$.

We define rewriting rules R to be mined from the graph sequences in the form of d_A as FTSs, each of which contains a subsequence of $seq(C'_{1,m'}(x))$ and TRs to transform $c_g'^{(m')}$ to g; i.e.,

$$R = \{s_1 \lozenge s_2 \mid s_1 \sqsubseteq seq(C'_{1,m'}(x)),\ s_2 \sqsubseteq seq(\langle c_g'^{(m')}, g\rangle),\ (x,g) \in D,\ sup(s_1 \lozenge s_2) \geq sup'\}, \quad (1)$$

where sup' and $s_1 \lozenge s_2$ are the minimum support threshold and the concatenation of transformation sequences s_1 and s_2, respectively. We refer to s_1 of r in R as the precondition of rewriting rule r. Another way of generating a graph sequence from the transition sequence is to append the correct state $c_g^{(k)}$ to its transition subsequence immediately after the parser selects the incorrect transition; i.e., $d_B = \langle c_g^{(1)}, \cdots, c_g^{(k-1)}, c_g'^{(k)}, c_g^{(k)}\rangle$. Then, similar to the first approach, rewriting rules are mined from the graph sequences in the form of d_B. Since $f \sqsubseteq seq(d_B) \Rightarrow f \sqsubseteq seq(d_A)$, all FTSs mined from graph sequences generated in the second approach are also mined from those in the first approach. However, the converse does not hold, since d_A contains information about vertices and edges that is not included in d_B. Since this information is not used in feature vectors characterizing $c_g^{(k-1)}$ to select the next transition at state $c_g^{(k-1)}$, this may explain why incorrect dependency graphs are returned by transition-based dependency parsers. In addition, the first approach may contain "maximal" information gathered from the whole dependency graph that is not available during transitions of the conventional parser (Attardi and Ciaramita, 2008). Therefore, we use the first approach to generate graph sequences. In addition, to distinguish TRs in s_1 of Eq. (1) from those in s_2, we assign label l_2 to edges in g that are not in $c_g'^{(m')}$, and label l_1 to all other edges.

Example 3 *Figure 8 shows a graph sequence generated by appending the correct final state g to the transition sequence for the sentence in Example 1, where the parser selects an incorrect transition from $c_g^{(5)}$ to $c_g'^{(6)}$. Since edge $(2,4)$ is not in $c_g'^{(6)}$ but is in g, label l_2 is assigned to the edge. The transformation sequence of the graph sequence is given as $\langle vi^{(0,1)}_{[0,\$]} vi^{(0,2)}_{[0,root]} vi^{(0,3)}_{[1,I]} vi^{(0,4)}_{[1,PRP]} vi^{(1,1)}_{[2,saw]} vi^{(1,2)}_{[2,VBD]} ei^{(2,1)}_{[(2,1),l_1]} vr^{(2,2)}_{[1,reduced]} ei^{(3,1)}_{[(0,2),l_1]} vi^{(3,2)}_{[3,him]} vi^{(3,3)}_{[3,PRP]} ei^{(4,1)}_{[(2,3),l_1]} vi^{(4,2)}_{[4,.]} vi^{(4,3)}_{[4,`.']} ed^{(5,1)}_{[(2,3),\bullet]} ei^{(5,2)}_{[(2,4),l_2]}\rangle$, where ROOT, PRP, VBD, and '.' are the parts of speech (POSs) of the corresponding words*[1].

Let $r = \langle vi^{(0,1)}_{[0,root]} vi^{(1,1)}_{[2,VBD]} ei^{(3,1)}_{[(0,2),l_1]}\ ei^{(4,1)}_{[(2,3),l_1]} vi^{(4,2)}_{[4,.]} ei^{(5,1)}_{[(2,4),l_2]}\rangle$ be a rewriting rule. If the parser has the rewriting rule r and is in state $c_g'^{(6)}$ in Example 3, the method proposed in this paper deletes edge $(3,4)$ from $c_g'^{(6)}$, and adds edge $(2,4)$ to $c_g'^{(6)}$, by applying r to transit to another state g in Fig. 4 that can reach the correct final state, since the transformation sequence of transition sequence $\langle c_g^{(1)}, \cdots, c_g^{(5)}, c_g'^{(6)}\rangle$ contains the precondition of r as a subsequence. Therefore, the

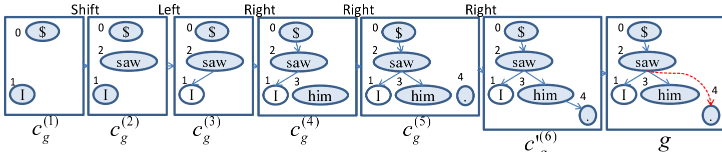


Figure 8: Graph sequence for a transition sequence

[1]We have *a priori* knowledge that each vertex in a state has at most one parent. Therefore, the fact that a TR t for inserting an edge labeled l_2 exists in transformation sequence s indicates that another TR for deleting an edge whose dependent is identical to t must exist in s. For this reason, in our implementation, we do not include TRs for deleting edges in s to reduce the computation time of GTRACE, which increases exponentially with the average length of the transformation sequences in its input.

rewriting rule that transforms from ${c'_g}^{(6)}$ to g corresponds to a bypass between states in the search space.

GTRACE mines a vast set of FTSs, R, in Eq. (1) from the graph sequences. Next, we discuss how to select certain FTSs from these. Desirable rewriting rules are those that rewrite states reaching incorrect final states to those reaching the correct final states. By rewriting states, the attachment score for a parser using rewriting rules should be better than that for a parser without rewriting rules. This is achieved by selecting the rewriting rule satisfying the following equation.

$$r = \arg\max_{r \in R} \left[\frac{1}{|D|} \sum_{x \in D} \#p(S_r, x) - \#p(S, x) \right], \tag{2}$$

where S_r and S are transition-based parsers with and without rewriting rule r, and $\#p$ is the number of words with correct parents in the dependency graph returned by the parser for sentence x. If $r = s_1 \Diamond s_2$ is a rule in R as given in Eq. (1), $|D| \times \sup(s_1 \Diamond s_2)$ and $|D| \times \big(\sup(s_1) - \sup(s_1 \Diamond s_2)\big)$ are the expected numbers of sentences in D, correctly and incorrectly rewritten by r, respectively. Thus, we obtain the following approximation.

$$\frac{1}{|D|} \sum_{x \in D} \#p(S_r, x) - \#p(S, x) \simeq \sup(s_1 \Diamond s_2) - \big(\sup(s_1) - \sup(s_1 \Diamond s_2)\big) = 2 \sup(s_1 \Diamond s_2) - \sup(s_1). \tag{3}$$

We select certain FTSs that maximize the right-hand side of Eq. (3) from R. In addition, we limit the mined rewriting rules such that the number of TRs in s_2 is 1, which is denoted as $|s_2| = 1$, for the following reason. If there is a rewriting rule $r = s_1 \Diamond s_2$ that correctly rewrites a state c into c', where $s_2 = \langle tr_1 tr_2 \rangle$ consists of two TRs, we divide r into $r_1 = s_1 \Diamond tr_1$ and $r_2 = s'_1 \Diamond tr_2$, where $s'_1 = s_1 \Diamond tr_1$. If r_1 rewrites a state c into another state c'', the state c'' is rewritten into c' by r_2. Even if we limit the mined rewriting rules such that $|s_2| = 1$, we can mine r_1 and r_2, and the state c can be rewritten correctly by r_1 and r_2. In addition, by the limitation, we efficiently mine all rewriting rules because mining rewriting rules is a combinatorial problem of TRs.

We propose a method for mining rewriting rules from transition sequences traversed by a dependency parser. For the sake of simplicity of explanation, we first explain the basic algorithm for mining rewriting rules from transition sequences generated by a transition-based parser with beam width 1, and then expand it using a dependency parser incorporating the beam search principle. The left part of Fig. 9 gives the pseudo-code for mining the set of rewriting rules R from the transition sequences. Let D be a corpus $D = \{(x, g)\}$ consisting of sentences $x = \langle w_0, w_1, \cdots, w_n \rangle$ and their dependency graphs g. In line 6a, ParseWithRules returns a transition sequence d by parsing sentence x using the rewriting rules R. Next, in line 7a, after appending g to the tail of d, denoted by $\langle d \Diamond g \rangle$, $\langle d \Diamond g \rangle$ is added to DB. Subsequently, in line 8a, the attachment score is updated after comparing the final state $c_g^{(m)}$ with the correct dependency g of sentence x. In line 9a, if the attachment score for $R \cup \{r\}$ is not greater than that for R, R is returned. Otherwise, r is added to R. In line 13a, a rewriting rule r satisfying Eqs. (2) and (3) is mined from the FTSs enumerated by GTRACE from DB under the minimum support threshold sup'.

The right part of Fig. 9 gives the pseudo-code for parsing sentence x using the rewriting rules R to return a transition sequence for x. The procedures, except for those in lines b6 to b13, are similar to those in Fig. 2. In line b5, the last state in the transition sequence d is substituted

RuleMiner(D, sup')

1a) $R \leftarrow \emptyset$
2a) $r \leftarrow null$
3a) while
4a) $DB \leftarrow \emptyset$
5a) for sentence $(x = \langle w_0, \cdots, w_n \rangle, g) \in D$
6a) $d \leftarrow$ ParseWithRules $(x, R \cup \{r\})$, where $d = \langle c_g^{(1)}, \cdots, c_g^{(m)} \rangle$
7a) $DB \leftarrow DB \cup \{\langle d \Diamond g \rangle\}$
8a) $evaluate(c_g^{(m)}, g)$
9a) if $R \neq \emptyset$ and the attachment score is saturated,
10a) return R
11a) if $r \neq null$
12a) $R \leftarrow R \cup \{r\}$
13a) $r \leftarrow MineRewritingRule(DB, sup')$
14a) if $r = null$
15a) return R

ParseWithRules$(x = \langle w_0, \cdots, w_n \rangle, R)$

1b) $C_{andidates} \leftarrow \{\langle c_s(x) \rangle\}$
2b) while $\{last(d) \mid d \in C_{andidates}\} \nsubseteq C_F$
3b) $A_{genda} \leftarrow \emptyset$
4b) for each $d \in C_{andidates}$
5b) $c_g = (N, A, N') \leftarrow last(d)$
6b) for each $r = s_1 \Diamond s_2 \in R$
7b) $(a, b) \leftarrow (v_1, v_2)$ $s.t.$ $s_2 = ei^{(j)}_{[(v_1, v_2), l_2]}$
8b) if $s_1 \sqsubseteq seq(d)$, where $\phi : ID(s_1) \rightarrow ID(seq(d))$
9b) $(i, j) \leftarrow (\phi(a), \phi(b))$
10b) $c_g \leftarrow (N, A \cup \{(i, j), N'\})$
11b) if $\exists i'$ s.t. $(i', j) \in A \wedge i' \neq i$
12b) $c_g \leftarrow (N, A \setminus \{(i', j), N'\})$
13b) $d \leftarrow \langle d \Diamond c_g \rangle$
14b) $A_{genda} \leftarrow A_{genda} \cup \{\langle d \Diamond t(c_g) \rangle \mid t \in \{\text{Left, Right, Reduce, Shift}\}\}$
15b) $C_{andidates} \leftarrow best(A_{genda})$
16b) return $C_{andidates}$

Figure 9: Algorithms for mining rewriting rules and parsing using the rules (beam width is 1)

into c_g by function $last$. All possible transition sequences for c_g are generated in line 14b, and the best of these is selected by function $best$ in line 14b. If there is a rewriting rule whose precondition is contained in $seq(d)$ and its mapping ϕ from vertex IDs in the precondition of r to vertex IDs in $seq(d)$, state $c_g = (N, A, N')$ is rewritten in line 10b or 12b and the parser transits to another state. In line 10b, edge (i, j) corresponding to (a, b), is added to A. In addition, if the j-th word has another parent i' different from i, edge (i', j) is deleted from A in line 12b.

In the case of a parser using a beam search with width b, ParseWithRules returns a set of transition sequences $=\{d_i \mid i = 1, \ldots, b\}$, instead of a single transition sequence. We assume that d_1 is the transition sequence whose final state has the best score of all $\{d_i\}$. If the final state of d_1 is isomorphic to g, we do not append g to d_i for any i, because we do not need any rewriting rules to transform states in transition sequences for sentences for which the parser returns correct final states. Otherwise, after appending g to the tail of each d_i, denoted by $\langle d_i \Diamond g \rangle$, $\langle d_i \Diamond g \rangle$ is added to DB similarly to the case in line a7. Therefore, DB consists of $|D| \times b$ graph sequences. On the other hand, the code for ParseWithRules incorporating the beam search principle is almost the same as the original. In line 15b, the b best transition sequences are selected from A_{genda} by function $best$.

5 Experiments

We evaluated the proposed method using English Penn Treebank data. We used Yamada's head rules to convert the phrase structure to a dependency structure (Yamada and Matsumoto, 2003). We also used the averaged perceptron algorithm with early-update strategy (Collins and Roark, 2004), where weights are updated whenever the gold-standard action-sequence falls off the beam, while the rest of the sequence is ignored. The idea behind this strategy is that later mistakes are often caused by earlier ones, and are irrelevant if the parser is already on the wrong track (Huang and Sagae, 2010).

We split the Wall Street Journal part of the corpus into sections 02-11 for training, sections

12-21 for mining rewriting rules, and section 22 for development. The set of feature templates in Zhang and Clark (2008) characterizing states in the parser was used. Figures 10, 13, and 14 show three of the few dozen rewriting rules mined using the proposed method under a minimum support threshold of 0.05% and beam width of 4, where h, i, j, k, and l are word IDs satisfying $h < i < j < k < l$, and the terms in each circle are words or their POS-tags. The minimum support threshold was set through trial and error using the development data. The supports $sup(s_1)$ and $sup(s_1 \Diamond s_2)$ of the rule shown in Fig. 10 are 1.22% and 0.95%, respectively. In addition, Fig. 11 shows some of the sentences in the corpus whose transition sequences are correctly rewritten by the rule. In Fig. 11, the annotations i, j, and h after words correspond to the respective word IDs in Fig. 10. Here, we explain the rule using concrete examples. The two sentences

- "$ Dozens of workers were injured." and
- "$ Dozens of workers were injured, authorities said."

and their correct dependency graphs, are shown on the left and right sides of Fig. 12, respectively. If the parser is in state $c_g^{(6)}$, where $\sigma = [0]$ and $\beta = [4, 5, \cdots, n]$, when parsing the first sentence, it usually selects Right to transit to the next state. Similarly, the parser incorrectly selects Right, instead of Shift, to transit to the next state from $c_g^{(6)}$ when parsing the second sentence. This is because when the parser is in state $c_g^{(6)}$, it does not know whether the phrase "authorities said." occurs at the end of the sentence. As mentioned in Section 2, if the parser shown in Fig. 2 adds an incorrect dependency between words once, it never reaches the correct final state. This rule rewrites the fifth state $g^{(5)}$ by deleting edge (i, j) and adding edge (h, j) without backtracking. Since we limit the mined rewriting rules to $r = s_1 \Diamond s_2$ where $|s_2| = 1$, this rule does not include deleting edge (h, i) and adding edge (j, i) in $g^{(5)}$ in the rewrite. However, we obtained another rewriting rule to do this.

The rewriting rule shown in Fig. 13 suggests that our parser incorrectly parses sentences containing "because of NN". The supports, $sup(s_1)$ and $sup(s_1 \Diamond s_2)$, of the rule are 0.71% and 0.54%, respectively. We assume that the parser is in state $(\sigma, \beta, A) = ([\cdots, h, i], [j, \cdots], A)$, where w_h ="because", w_i ="of", the POS-tag for w_j is NN (noun, singular or mass), and $(h, i) \in A$. Since the parser using the feature templates in Zhang and Clark (2008) does not know what the word for the parent of the stack top is, the parser incorrectly selects Right, instead of Reduce, to transit to the next state. This rule rewrites state $g^{(5)}$ by deleting edge (i, j) and adding edge (h, j), after the parser adds an incorrect edge.

The supports, $sup(s_1)$ and $sup(s_1 \Diamond s_2)$, of the rule shown in Fig. 14 are 0.053%. After mining the rewriting rule, we investigated words corresponding to vertices h, i, and j by scanning the corpus. The words, (w_h, w_i, w_j), were either (Procter, &, Gamble), (Peabody, &, Co.), (Ogilvy, &, Mather), (Young, &, Rubicam), (Shea, &, Gould), (Standard, &, Poor), (Bausch, &, Lomb), or (Dun, &, Bradstreet). The trigram of words comprising each triplet is a compound proper noun. The rule suggests incorrect parsing of words consisting of proper compound nouns. Since the parser does not know that the triplets are proper compound nouns, it cannot

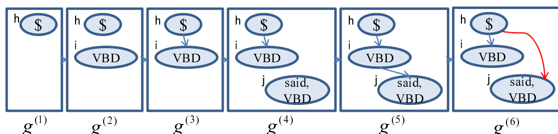


Figure 10: Mined rule #1

$(h) Dozens of workers were(i) injured, authorities said(j).
$(h) But the Fed move was(i) a small gesture, traders said(j).
$(h) Mr. Agnos declined(i) the invitations, the White House said(j).
$(h) They continued(i) to represent that to the board," said(j) Mr. Lloyd.
$(h) Some laggard food issues attracted(i) bargain-hunters, traders said(j).

Figure 11: Sentences for which the rule in Fig. 10 correctly rewrites transition sequences

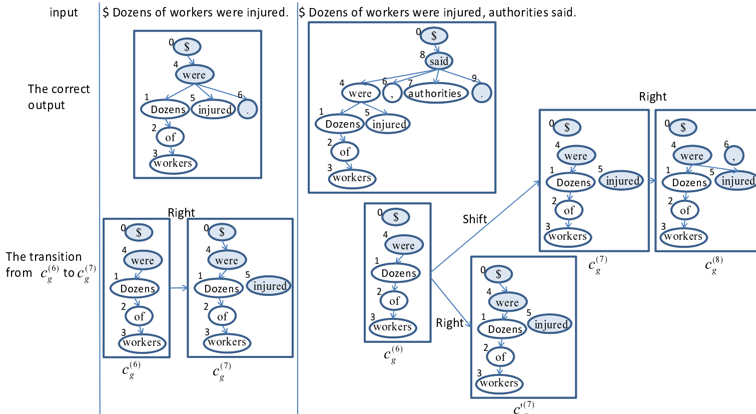


Figure 12: Transition of two similar sentences

correctly parse sentences containing "A & B's NN", where "A & B" is a proper compound noun. This rule rewrites state $g^{(9)}$ by deleting edge (l, j) and adding edge (i, j).

As shown above, the proposed method has the benefits that the rules mined by the method are human-readable and easily understandable. In addition, the rewriting rules contain context that is more complex and detailed than a set of features of the conventional parser, because of the use of the graph representation. Furthermore, if the mined rules are valid grammatically, and a dependency structure obtained by the proposed method, after being rewritten by the rules, is different from a dependency structure in the corpus made by humans, the latter dependency structure may contain incorrect dependencies. The proposed method is therefore also useful for rectifying human errors in the corpus.

According to the knowledge obtained from the mined rewriting rules, we designed three new types of feature templates characterizing states in the parser. First, according to the knowledge obtained from the first rewriting rule, we added feature templates N0w∘Flag and N0t∘Flag, where N0w and N0t denote the word at the top of the stack and its POS-tag in the parser, respectively, Flag is either true or false, and N0w∘Flag and N0t∘Flag are their concatenations. Flag is true, if one of the words "said", "say", or "says" appears after N0w in the sentence. Flag can be calculated in time linear to the length of the sentence during preprocessing. Since features of these templates provide the parser with information on whether a sentence contains "said", "say" or "says" at the end, the parser has a high probability of correctly parsing such sentences. Second, according to knowledge obtained from the second rewriting rule, we added feature templates SPTw∘STw∘N0t and SPTw∘STt∘N0t, where STw, STt, and STPw are the word at the top of the stack, its POS tag, and the word at the parent of the stack top, respectively. The

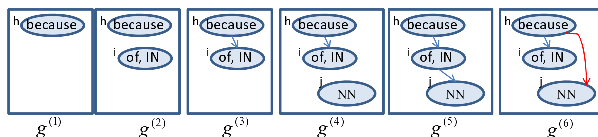


Figure 13: Mined rule #2

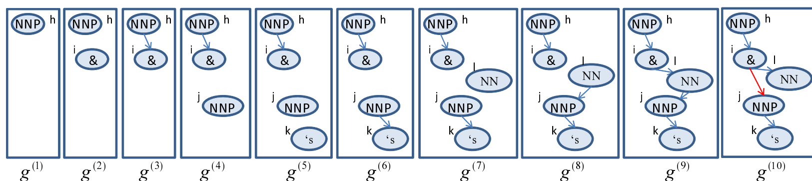


Figure 14: Mined rule #3

arc-eager shift -reduce parser using these feature templates has word information on the parent of the stack top, and is expected to parse sentences containing "because of NN" correctly. Third, according to knowledge obtained from the third rewriting rule, we added feature templates N-2w∘N-1w∘N0w, N-1w∘N0w∘N1w, and N0w∘N1w∘N2w, where N0w is the word at the buffer top, and N-2w, N-1w, N1w, and N2w are words before and after that word. Therefore, N-2w∘N-1w∘N0w, N-1w∘N0w∘N1w, and N0w∘N1w∘N2w are trigrams containing N0w in a sentence. The arc-eager shift-reduce parser using these feature templates is expected to parse sentences containing proper compound nouns correctly.

Figure 15 shows (1) the attachment scores for an arc-eager shift-reduce parser using only the conventional feature templates in Zhang and Clark (2008), (2) that using the feature templates given above as well as the conventional feature templates, and (3) that using the feature templates given above as well as the conventional feature templates and mined rewriting rules[2] for various beam widths. In these experiments, we split the WSJ part of the corpus into sections 02-21 for training, section 22 for development, and section 23 for testing. The figure shows that attachment scores are improved by adding the new feature templates to the conventional feature templates. In particular, the degree of improvement is large when the beam width is small. Although we used knowledge obtained from the three rewriting rules, greater improvement is expected by mining more rewriting rules and designing more feature templates based on the knowledge obtained from these rules. Table 2 reproduces the attachment scores for various dependency parsers given in Hayashi et al. (2012) including those of our proposed method. The table shows that our method is comparable to parsers of the latest studies with respect to the attachment score.

6 Conclusion

This paper proposed a method for mining rewriting rules from transition sequences of an arc-eager dependency parser incorporating the beam search principle for English sentences. The rewriting rules mined by the proposed method are human-readable, and it is possible for us to design new parsers and to generate feature templates for the machine learner of the parser to avoid producing incorrect dependency graphs. In this study, we used GTRACE to analyze transition sequences, although there are other data structures for representing graph sequences,

[2]Result using rewriting rules with beam with 64 was not obtainable due to memory overflow.

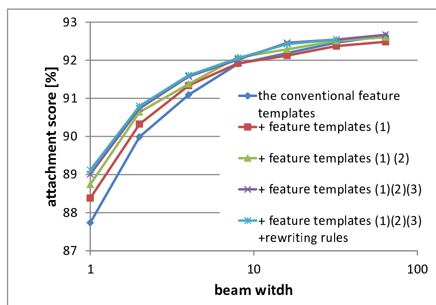


Figure 15: Attachment scores of arc-eager shift-reduce parsers using various feature templates

Table 2: Attachment scores for various methods

method	attachment score [%]
McDonald (McDonald and Pereira, 2006)	91.5
Koo (Koo and Collins, 2010)	93.04
Hayashi (Hayashi et al., 2011)	92.89
Goldberg (Goldberg and Elhadad, 2010)	89.7
Kitagawa (Kitagawa and Tanaka-Ishii, 2010)	91.3
Zhang (Sh beam 64) (Zhang and Clark, 2008)	91.4
Zhang (Sh+Graph beam 64) (Zhang and Clark, 2008)	92.1
Huang (beam+DP) (Huang and Sagae, 2010)	92.1
Zhang (beam 64) (Zhang and Nivre, 2011)	93.07
Hayashi (beam 32+pred 5+DP) (Hayashi et al., 2012)	92.5
Hayashi (beam 32+pred 5+DP+FIRST) (Hayashi et al., 2012)	92.6
Our method (Sh beam 64+additional features)	92.67

such as dynamic graphs (Borgwardt et al., 2006) and evolving graphs (Berlingerio et al., 2009), and algorithms for mining the graphs. Since insertions of vertices cannot be represented by dynamic graphs, and a vertex in an evolving graph always comes with an edge connected to the vertex, these data structures cannot be used to analyze transition sequences in transition-based parsers to mine rewriting rules. Compared with dynamic graphs and evolving graphs, the class of graph sequences is, therefore, general enough to apply to the analysis of transition sequences.

Revision rules proposed in Ahmed and Karypis (2011) transform only edges in dependency structures output by the dependency parser in post-processing. Since it is assumed that revision rules do not remove or add any vertices in the dependency structures, the revision rules cannot be applied to phrase structure analysis. On the other hand, rewriting rules transform states in the transition-based dependency parser. Since the method proposed in this paper can basically be applied to any transition systems whose internal states are represented by graphs, it can be applied to the phrase structure analysis. In addition, rewriting rules are more human-readable than revision rules. Kudo et al. (2005) proposed a method for extracting features represented by trees that are human-readable to obtain high attachment scores. Using the features, the transition-based parser selects a correct transition in each state. Compared with that method, using our method proposed in this paper, we obtain knowledge about why incorrect dependency graphs are returned by transition-based dependency parsers and knowledge about how we transform incorrect states into correct states.

References

Agrawal, R., and Srikant, R. 1994. Fast Algorithms for Mining Association Rules in Large Databases. *Proc. of Int'l Conf. on Very Large Data Bases (VLDB)*, 487–499.

Ahmed, R., and Karypis, G. 2011. Algorithms for Mining the Evolution of Conserved Relational States in Dynamic Networks. *Proc. of IEEE Int'l Conf. on Data Mining (ICDM)*, 1–10.

Attardi, G. and Ciaramita, M. 2008. Tree Revision Learning for Dependency Parsing. *Proc. of Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, (HLT-NAACL)*, 388–395.

Berlingerio, M., et al., 2009. Mining Graph Evolution Rules. *Proc. of Euro. Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, 115–130.

Borgwardt, K. M., et al., 2006. Pattern Mining in Frequent Dynamic Subgraphs. *Proc. of IEEE Int'l Conf. on Data Mining (ICDM)*, 818–822.

Collins, M., and Roark, B. 2004. Incremental parsing with the perceptron algorithm. *Proc. of the 42nd Annual Meeting of the Association for Computational Linguistics(ACL)*, pp. 111-118.

Culotta, A., and Sorensen, J. 2004. Dependency Tree Kernels for Relation Extraction. *Proc. of Annual Meeting of Association for Comp. Linguistics (ACL)*, 423–429.

Ding, Y., and Palmer, M. 2004. Automatic Learning of Parallel Dependency Treelet Pairs. *Proc. of Int'l Joint Conf. on Natural Language Processing*, 233–243.

Garey, M., and Johnson, D. 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York.

Goldberg, Y. and Elhadad, M. 2010. An efficient algorithm for easy-first non-directional dependency parsing. *Proc of Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics* (HLT-NAACL), pp. 742–750.

Hayashi, K. et al., 2011. The third-order variational reranking on packed-shared dependency forests. *Proc. of Conference on Empirical Methods in Natural Language Processing* (EMNLP), pp. 1479–1488.

Hayashi, K., et al., 2012. Head-driven Transition-based Parsing with Top-down Prediction. *Proc. of the Annual Meeting of the Association for Computational Linguistics(ACL)*.

Huang, L., and Sagae, K. 2010. Dynamic Programming for Linear-Time Incremental Parsing. *Proc. of Annual Meeting of the Association for Computational Linguistics (ACL)*, 1077-1086.

Inokuchi A., and Washio, T. 2008. A Fast Method to Mine Frequent Subsequences from Graph Sequence Data. *Proc. of IEEE Int'l Conf. on Data Mining (ICDM)*, 303–312.

Inokuchi, A., et al., 2012. Efficient Graph Sequence Mining Using Reverse Search. *IEICE Transactions* 95-D(7), 1947–1958

Inokuchi, A., et al., 2012. Mining Rules for Rewriting States in a Transition-Based Dependency Parser. *Proc of Pacific Rim International Conference on Artificial Intelligence* (PRICAI), 133–145.

Iwatate, M., et al., 2008. Japanese Dependency Parsing Using a Tournament Model. *Proc. of Int'l Conf. on Comp. Linguistics (COLING)*, 361–368.

Kitagawa, K. and Tanaka-Ishii, K. 2010. Tree-based deterministic dependency parsing - an application to Nivre's method. *Proc. of Annual Meeting of the Association for Computational Linguistics* (ACL), Short Papers, pp. 189–193.

Koo, T. and Collins, M. 2010. Efficient third-order dependency parsers. *Proc. of Annual Meeting of the Association for Computational Linguistics* (ACL), pp. 1–11.

Kubler, S., et al., 2009.*Dependency Parsing*. Morgan and Claypool Publishers.

Kudo, T., and Matsumoto, Y. 2002. Japanese Dependency Analysis using Cascaded Chunking. *Proc. of Conf. on Comp. Natural Language Learning (CoNLL)*, 63–69.

Kudo, T. et al. 2005. Boosting-based Parse Reranking with Subtree Features. *Proc. of Annual Meeting of the Association for Computational Linguistics (ACL)*.

Makinen, E. 1989. On the Subtree Isomorphism Problem for Ordered Trees. *Information Processing Letters* 32(5), 271–273.

McDonald, R. and Pereira, F. 2006. Online learning of approximate dependency parsing algorithms. *Proc. of Conference of European Chapter of the Association for Computational Linguistics* (EACL), pp. 81–88.

Nivre, J. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Comp. Linguistics* 34(4), 513–553.

Pei, J., et al., 2001. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2–6.

Yamada, H., and Matsumoto, M. 2003. Statistical dependency analysis with support vector machines. *Proc. of the International Workshop on Parsing Technologies (IWPT)*, pp 195–206.

Zhang, Y., and Clark, S. 2008. A Tale of Two Parsers: Investigating and Combining Graph-based and Transition-based Dependency Parsing. *Proc. of Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, 562-571.

Zhang, Y. and Nivre, J. 2011. Transition-based dependency parsing with rich non-local features. *Proc. of Annual Meeting of the Association for Computational Linguistics* (ACL), pp. 188–193.