Mining Rules for Rewriting States in a Transition-Based Dependency Parser

Akihiro Inokuchi[1], Ayumu Yamaoka[1], Takashi Washio[1], Yuji Matsumoto[2], Masayuki Asahara[3], Masakazu Iwatate[4], and Hideto Kazawa[5]

[1] Institute of Scientific and Industrial Research, Osaka University
[2] Nara Institute of Science and Technology
[3] National Institute for Japanese Language and Linguistics
[4] HDE, Inc.
[5] Google, Inc.

Abstract. Methods for mining graph sequences have recently attracted considerable interest from researchers in the data-mining field. A graph sequence is one of the data structures that represent changing networks. The objective of graph sequence mining is to enumerate common changing patterns appearing more frequently than a given threshold from graph sequences. Syntactic dependency analysis has been recognized as a basic process in natural language processing. In a transition-based parser for dependency analysis, a transition sequence can be represented by a graph sequence where each graph, vertex, and edge respectively correspond to a state, word, and dependency. In this paper, we propose a method for mining rules for rewriting states reaching incorrect final states to states reaching the correct final state, and propose a dependency parser that uses rewriting rules. The proposed parser is comparable to conventional dependency parsers in terms of computational complexity.

1 Introduction

Data mining is used to mine useful knowledge from large amounts of data. Recently, methods for mining graph sequences (dynamic graphs [4] or evolving graphs [3]) have attracted considerable interest from researchers in the data-mining field [9]. For example, human networks can be represented by a graph where each vertex and edge respectively correspond to a human and relationship in the network. If a human joins or leaves the network, the numbers of vertices and edges in the graph increase or decrease. A graph sequence is one of the data structures used to represent a changing network. Figure 1(a) shows a graph sequence that consists of four steps, five vertices, and edges among the vertices. The objective of graph sequence mining is to enumerate subgraph subsequence patterns, one of which is shown in Fig. 1(b), appearing more frequently than a given threshold from graph sequences.

Syntactic dependency parsing has been recognized as a basic process in natural language processing, and a number of studies have been reported [12,14,16,8].

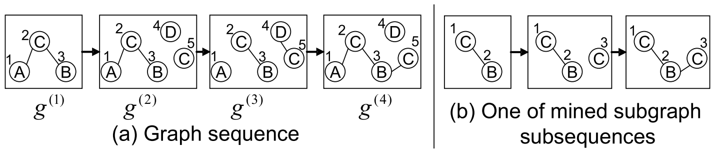

Fig. 1. Examples of a graph sequence and one of mined frequent patterns

One reason for its increasing popularity is the fact that dependency-based syntactic representations seem to be useful in many applications of language technology [11], such as machine translation [5] and information extraction [6]. In a transition-based dependency parser, a transition sequence can be represented by a graph sequence where each graph, vertex, and edge respectively correspond to a state, word, and dependency. Because of the nature of the algorithm where transition actions are selected deterministically, an incorrect selection of an action may adversely affect the remaining parsing actions. If characteristic patterns are mined from transition sequences for sentences analyzed incorrectly by a parser, it is possible to design new parsers and to generate better features in the machine learner in the parser to avoid incorrect dependency structures.

The first and main objective of this study is to demonstrate the usefulness of graph sequence mining in dependency analysis. Since methods for mining graph sequences were developed, they have been applied to social networks in Web services [3], article-citation networks [2], e-mail networks [4], and so on. In this paper, we demonstrate a novel application of graph sequence mining to dependency parsing in natural language processing. The second objective is to propose a method for mining rewriting rules that can shed light on why incorrect dependency structures are returned by transition-based dependency parsers. To mine such rules, the rules should be human-readable. If we identify the reason for incorrect dependency structures, it is possible to design new parsers and to generate better features in the machine learner in the parser to avoid incorrect dependency structures. The third objective is to propose a dependency parser that uses rewriting rules, where the method is comparable to conventional methods whose time complexity is linear with respect to the number of segments in a sentence. The fourth, but not a main, objective is to improve the attachment score, which is a measure of the percentage of segments that have the correct head, and the exact match score for measuring the percentage of completely and correctly parsed sentences.

2 Transition-Based Dependency Parsing

In this paper, we focus on dependency analysis using an "arc-standard parser" [14], which is a parser based on a transition system, for "Japanese sentences", for the sake of simplicity, because constraints in Japanese dependency structures are stronger than those in other languages. Japanese dependency structures have

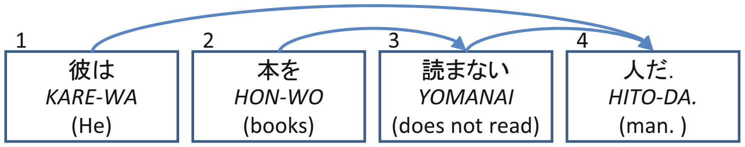

Fig. 2. Example of a Japanese sentence and its dependency structure

```
Parse($x = \langle w_1, w_2, \cdots, w_n \rangle$)
1)  $c \leftarrow c_s(x)$
2)  while $c \notin C_F$
3)      $c \leftarrow [o(c)](c)$
4)  return $c_{=}(N, A)$
```

Fig. 3. Dependency parser based on a transition system

Transitions	
Arc	$(N, A) \Rightarrow (N, A \cup \{(i, j)\})$ where $\{i, j\} = roots((N, A))$
Shift	$(N, A) \Rightarrow (N \cup \{\lvert N \rvert + 1\}, A)$
Preconditions	
Arc	c is not a tree, but a forest.
Shift	$\lvert N \rvert \neq n$

Fig. 4. Transitions of an arc-standard parser

strictly head-final, single-head, single-rooted, connected, acyclic, and projective constraints [10]. 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.

Most Japanese dependency parsers are based on bunsetsu segments (hereafter segments), which are a similar in concept to English base phrases.

Definition 1. *A dependency structure for a sentence* $x = \langle w_1, \cdots, w_n \rangle$ *consisting of* n *segments is represented as a directed rooted tree* $g = (V, E)$, *where* $V = \{1, \cdots, n\}$, $E \subset V \times V$, *and* n *is the root of the tree.* ■

Example 1. A dependency structure for a sentence $x = \langle$*KARE-WA, HON-WO, YOMANAI, HITO-DA.*$\rangle$ is represented by a directed graph without edge crossings, as shown in Fig. 2.

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

Definition 2. *A transition-based parser consists of* $S = (C, T, c_s, C_F)$, *where*
- $C = \{(N, A)\}$ *is a set of states, where* N *and* A *are subsets of* $V = \{1, \cdots, n\}$ *and* $N \times N$, *respectively,*
- T *is a set of transitions, where* $t \in T$ *is a partial function s.t.* $t : C \to C$,
- c_s *is an initial function satisfying* $c_s(x) = (\{1\}, \emptyset)$, *and*
- $C_F \subseteq C$ *is a set of final states, and* $c_F \in C_F$ *is a tree where* n *is the root.* ■

A transition sequence for $x = \langle w_1, \cdots, w_n \rangle$ 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 $(1 < i \leq m)$, $c_i = t(c_{i-1})$. We denote sets of vertices and edges for a state c as N_c and A_c, respectively.

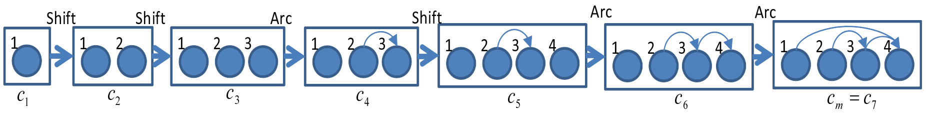

Fig. 5. Transition sequence for the sentence in Example 1

Definition 3. *A transition-based parser* $S = (C, T, c_s, C_F)$ *is incremental if* $N_c \subseteq N_{t(c)}$ *and* $A_c \subseteq A_{t(c)}$*.* ■

If a dependency parser is incremental, the numbers of vertices and edges in a state $c = (N, C)$ increase monotonically. In addition, the state $c = (N_c, A_c)$ is a forest that is a set of ordered trees, and a graph (N_c, A_c) is a subgraph of $c_m = (N_{c_m}, A_{c_m})$ that S returns.

Figure 3 shows the algorithm of a transition-based dependency parser. In Fig. 3, o is an oracle for selecting t to transit to the next state in a deterministic way. In particular, the arc-standard parser, which is a transition-based parser, selects either Arc or Shift to analyze Japanese sentences, as shown in Fig. 4, where *roots* returns a pair of the largest roots $\{i, j\}$ $(i < j)$ from a forest $c = (N, A)$[1]. If o selects Arc, then an edge (i, j) is added to transit from c to $t(c)$. Otherwise, the smallest vertex that does not exist in $c = (N, A)$ is added to c to transit from c to $t(c)$. Since o is a function for determining whether the i-th segment is the dependent of the j-th segment, it is implemented with a binary classifier, such as a support vector machine (SVM), for feature vectors that characterize the i-th and j-th segments [11].

Since the arc-standard parser is incremental, Arc is selected $n - 1$ times and Shift is also selected $n - 1$ times to reach the final state. The time complexity of the parser for a sentence with n segments is therefore $O(n\theta)$, where we assume o, which is a binary classifier, returns its output in at most θ time.

Example 2. Figure 5 shows a transition sequence from the initial state to the final state for the dependency structure shown in Fig. 2. The words are omitted because of a lack of space. In the sequence, Shift, Shift, Arc, Shift, Arc, and Arc are selected by o, in that order.

Figure 6 shows the search space for the sentence in Example 1. Since a search space consisting of states is a tree, there is only one transition sequence from the initial state to the correct final state. In addition, the branching factor of the tree is at most two. Since the function o selects a transition between two branches, if the function o selects an incorrect transition once, the parser never reaches the correct final state. A straightforward approach to avoiding this mistake is to integrate backtracking or a probabilistic algorithm with the parser. However, this impairs the advantages of a parser whose time complexity is linear with respect to the number of segments in a sentence.

[1] Although the arc-standard parser is defined using a stack and queue in many books and articles, in this paper, we define it using graphs to link dependency parsing to graph sequence mining.

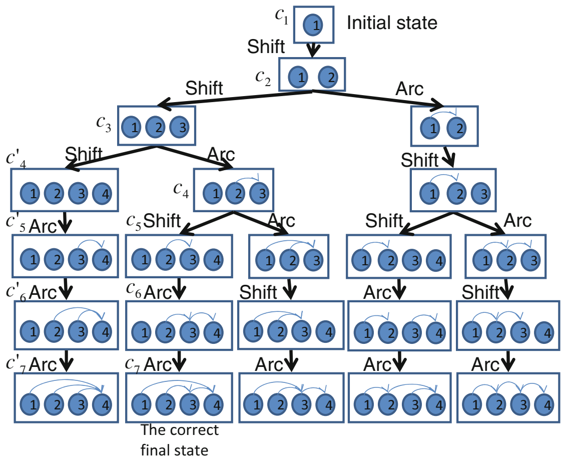

Fig. 6. Search space for the sentence in Example 1

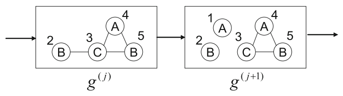

Fig. 7. Change between two successive graphs

In this paper, we propose a method for mining rules for rewriting from states reaching incorrect final states to states reaching the correct final state, and propose a dependency parser that uses rewriting rules. The rewriting rules correspond to bypasses among states in the search tree shown in Fig. 6, and the proposed parser is comparable to a conventional dependency parser in terms of computational complexity. To describe the proposed method, we explain another method, called 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. The 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 have proposed transformation rules for representing graph sequences compactly under the assumption that "the change is gradual" [9]. In other words, only a small part of the structure changes, while the other part remains unchanged between two successive graphs $g^{(j)}$ and $g^{(j+1)}$ in a graph sequence. For example, the change between two successive graphs $g^{(j)}$ and $g^{(j+1)}$ in the graph sequence shown in Fig. 7 is represented as an ordered list of two transformation rules $\langle vi^{(j)}_{[1,A]}, ed^{(j)}_{[(2,3),\bullet]}\rangle$. This list implies that a vertex with ID 1 and label A is inserted (vi), and then an edge between vertices with IDs 2 and 3 is deleted (ed). By assuming the change in each graph to be gradual, we can represent a graph sequence compactly, even if the graph in the graph sequence has many vertices and edges. We have also proposed a method, called GTRACE, for efficiently mining all frequent patterns from ordered lists of transformation

Table 1. TRs for representing graph sequence

Vertex Insertion $vi^{(j,k)}_{[u,l]}$	Insert a 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 a label of a 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 into $g^{(j,k)}$ 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 $g^{(j,k+1)}$.

rules. A transition sequence in the dependency parser is represented as a graph sequence. In addition, since the change between two successive graphs in the graph sequence is an addition of a vertex (Shift) or of an edge (Arc), 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 $l : V \cup E \to L$. In addition, a graph sequence is an ordered list of labeled graphs and is represented as $d = \langle g^{(1)}, \cdots, g^{(z)} \rangle$.

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

Definition 4. *The differences between the 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 the interpolations as $d = \langle d^{(1)}, \cdots, d^{(z-1)} \rangle$.* ■

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

Definition 5. *Let a 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 interpolation in which the edit distance between any two successive graphs is 1 and the edit distance between any two graphs is minimum.* ■

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

Definition 6. *A TR transforming $g^{(j,k)}$ to $g^{(j,k+1)}$ is represented by $tr^{(j,k)}_{[o_{jk},l_{jk}]}$, where*

- *tr is a transformation type that is either insertion, deletion, or relabeling of a vertex or an edge,*
- *o_{jk} is a 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 $tr^{(j,k)}_{[o,l]}$. We use six TRs in Table 1. In summary, we define a transformation sequence as follows.

Definition 7. *A graph sequence $d^{(j)} = \langle g^{(j,1)}, \cdots, g^{(j,m_j)} \rangle$ is represented by $s_d^{(j)} = \langle tr^{(j,1)}_{[o,l]}, \cdots, tr^{(j,m_j-1)}_{[o,l]} \rangle$. Moreover, a graph sequence $d = \langle g^{(1)}, \cdots, g^{(z)} \rangle$ is represented by a transformation sequence $s_d = \langle s_d^{(0)}, \cdots, s_d^{(z-1)} \rangle$.* ■

The notation of 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.

When a transformation sequence s'_d is a subsequence of a transformation sequence s_d, there is a mapping ϕ from vertex IDs in s'_d to those in s_d, and it is denoted as $s'_d \sqsubseteq s_d$. We omit its detailed definition because of a lack of space (see [9] for a detailed definition). Given a set of graph sequences $DB = \{\langle g^{(1)}, \cdots, g^{(z)} \rangle\}$, we define a support $\sigma(s_p)$ of a transformation sequence s_p as $\sigma(s_p) = |\{d \mid d \in DB, s_p \sqsubseteq s_d\}|/|DB|$, where s_d is a transformation sequence of d. We call a transformation sequence whose support is no less than the minimum support σ' a frequent transformation subsequence (FTS). Given a set of graph sequences, GTRACE efficiently enumerates a set of all FTSs from the set.

4 Mining Rules for Rewriting States

As mentioned in Section 2, if the parser shown in Fig. 3 selects the incorrect transition once, it never reaches the correct state. In this paper, we aim to discover rules for rewriting from states reaching incorrect final states to states reaching the correct final state. To discover these rewriting rules, we mine FTSs from graph sequences $\langle c_1, \cdots, c_m, g \rangle$, each of which consists of a transition sequence $\langle c_1, \cdots, c_m \rangle$ traversed by the parser and its correct dependency structure g. If $c_m = g$, then the final state c_m is correct and there are no TRs for transforming c_m into g. Otherwise, c_m is an incorrect final state and the TRs for transforming c_m into g are either

- transformation rules for inserting edges in g and not in c_m, or
- transformation rules for deleting edges in c_m and not in g.

As mentioned above, the rewriting rules to be mined are rules for transforming graphs in states that do not reach the correct dependency structure into

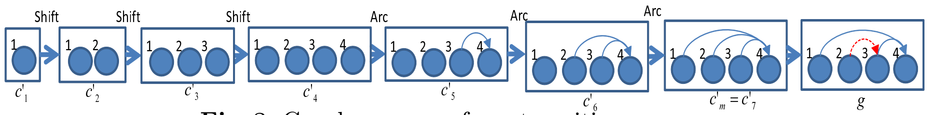

Fig. 8. Graph sequence for a transition sequence

graphs in other states that reach the correct structure for many sentences. The rewriting rules are therefore FTSs containing TRs for transforming c_m into g. To distinguish TRs for transforming c_m to g from other rules, we assign a label l_2 to edges in g and not in c_m, and a label l_1 to the other edges.

Example 3. Figure 8 shows a graph sequence d_A generated by appending the correct dependency structure g to the transition sequence for the sentence in Example 1, where the function o in a parser selects an incorrect transition from c'_3 to c'_4. Since the edge $(2,3)$ is not in c'_m and is in g, the label l_2 is assigned to the edge. The transformation sequence of the graph sequence is given as $s_{d_A} = \langle vi^{(0,1)}_{[1]} vi^{(1,1)}_{[2]} vi^{(2,1)}_{[3]} vi^{(3,1)}_{[4]} ei^{(4,1)}_{[(3,4),l_1]} ei^{(5,1)}_{[(2,4),l_1]} ei^{(6,1)}_{[(1,4),l_1]} ei^{(7,1)}_{[(2,3),l_2]} \rangle$[2][3].

We select an FTS whose confidence is the highest among mined FTSs whose last TR is the edge insertion of label l_2. The definition of the confidence of an FTS is similar to basket analysis [1], as described in the following. We call the selected FTS a rewriting rule.

Definition 8. *Given an FTS s, let s' be the prefix of s, obtained by removing the last TR in s. The confidence of s is defined as $\sigma(s')/\sigma(s)$, and s' is called a body of s. In addition, a function for returning an edge to which the last TR in s is applied is defined as $lastEdge(s)$.* ■

Example 4. When $r = \langle vi^{(0,1)}_{[2]} vi^{(1,1)}_{[3]} vi^{(2,1)}_{[4]} ei^{(3,1)}_{[(2,4),l_1]} ei^{(4,2)}_{[(2,3),l_2]} \rangle$ is a rewriting rule, its confidence is $\sigma(\langle vi^{(0,1)}_{[2]} vi^{(1,1)}_{[3]} vi^{(2,1)}_{[4]} ei^{(3,1)}_{[(2,4),l_1]} \rangle)/\sigma(r)$, and $lastEdge(r) = (2,3)$.

If a parser has the rewriting rule r of Example 4 and is in the state c'_6 of Example 3, the method proposed in this paper adds an edge $(2,3)$ to c'_6, and deletes an edge $(2,4)$ from c'_6, by applying r to transit another state c_6 in Fig. 5 that can reach the correct final state, since the transformation sequence of a transition sequence $\langle c'_1, \cdots, c'_6 \rangle$ contains the body of r as a subsequence. Therefore, the rewriting rule corresponds to a bypass from c'_6 to c_6 in the search tree shown in Fig. 6.

[2] Although each vertex is labeled by information such as words and parts-of-speech (POSs), the labeling depends on the features generated for a binary classifier in a parser. The details of labeling vertices are discussed in Section 5.

[3] 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 with l_2 exists in a transformation sequence s indicates that another TR for deleting an edge whose dependent is identical to t must exist in s. For this reason, we do not include TRs for deleting edges in s to reduce the computation time of GTRACE, which exponentially increases with the average length of the transformation sequences in its input.

```
    **RuleMiner**$(D, \sigma')${
 1) $R \leftarrow \emptyset$
 2) $r \leftarrow null$
 3) while
 4)   $DB \leftarrow \emptyset$
 5)   for sentence $(x = \langle w_1, \cdots, w_n \rangle, g) \in D$
 6)     $d \leftarrow$ ParseWithRules $(x, R \cup \{r\})$
 7)     $DB \leftarrow DB \cup \{d \Diamond g\}$
 8)     $evaluete(c_m, g)$, where $d = \langle c_1, \cdots, c_m \rangle$
 9) if $R \neq \emptyset$ and the attachment
        score is saturated.
10)       return $R$
11)   if $r \neq null$
12)     $R \leftarrow R \cup \{r\}$
13)   $r \leftarrow MineRewritingRule(DB, \sigma')$
14)   if $r = null$
15)       return $R$
```

```
    **ParseWithRules**$(x = \langle w_1, \cdots, w_n \rangle, R)$
 1) $c \leftarrow c_s(x)$
 2) $d \leftarrow \langle c \rangle$
 3) while $c \notin C_F$
 4)   $c \leftarrow [o(c)](c)$
 5)   $d \leftarrow d \Diamond c$
 6)   $s_d \leftarrow$the transformation sequence of $d$
 7)   for $r \in R$
 8)     $(a, b) \leftarrow lastEdge(r)$
 9)     if $body(r) \sqsubseteq s_d$,
          where $\phi : ID(body(r)) \rightarrow ID(s_d)$
10)       $(i, j) \leftarrow (\phi(a), \phi(b))$
11)       $c \leftarrow (N_c, A_c \cup \{(i, j)\})$
12)       if $\exists j'$ s.t. $(i, j') \in A_c \wedge j' \neq j$
13)         $c \leftarrow (N_c, A_c \setminus \{(i, j')\})$
14)       $d \leftarrow d \Diamond c$
15) return $d$
```

Fig. 9. Algorithms for mining rewriting rules and for parsing with the rules

Another way to generate a graph sequence from a transition sequence is to append to the correct state to the transition sequence immediately after the oracle in the parser selects the incorrect transition. In the case of Example 3, the graph sequence generated in this way is $d_B = \langle c'_1, \cdots, c'_4, c_4 \rangle$, where c_4 is of Fig. 5. Any subsequence of the transformation sequence s_{d_B} of d_B is always a subsequence of s_{d_A} in Example 3. In addition, d_A contains the information about vertices and edges that are not contained in d_B. Therefore, we use the approach to generate graph sequences in the form of d_A. Similarly, if r is a subsequence of s_{d_B}, r is a subsequence of s_{d_A}. Therefore, we apply r to c'_6 that is not the final state.

We propose a method for mining rewriting rules from transition sequences traversed by a dependency parser. The left part of Fig. 9 shows the pseudo-code for mining a set of rewriting rules R from the transition sequences. Let D be a corpus $D = \{(x, g)\}$ consisting of tokenized sentences $x = \langle w_1, \cdots, w_n \rangle$ and their dependency structures g. In Line 6, ParseWithRules returns a transition sequence d by parsing a sentence x using rewriting rules R. Next, in Line 7, after appending g to the tail of d, which is denoted by $d \Diamond g$, $d \Diamond g$ is added to DB. Subsequently, in Line 8, the attachment score is updated after comparing the final state c_m with the correct dependency g of the sentence x. In Line 9, if the attachment score for $R \cup \{r\}$ is no greater than that for R, then R is returned. Otherwise, r is added to R. In Line 13, a rewriting rule r with the highest confidence is mined among the FTSs enumerated by GTRACE from DB under the minimum support threshold σ'.

The right part of Fig. 9 shows the pseudo-code for parsing a sentence x using rewriting rules R to return a transition sequence for x. The procedures from Line 1 to Line 5 are similar to those in Fig. 3. If there is a rewriting rule whose body is contained in s_d and its mapping ϕ from vertex IDs in the body of r to vertex IDs in s_d, the state c is rewritten in Line 11 or 13 and is transited to

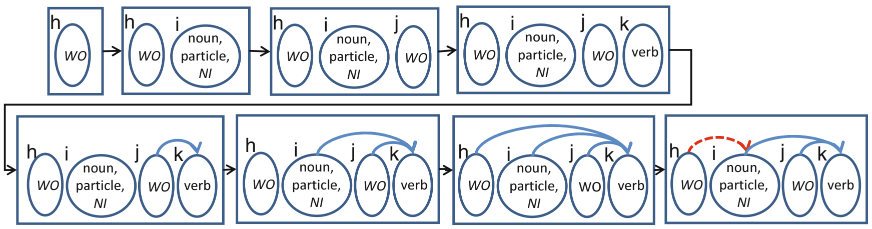

Fig. 10. One of the mined rules in the first loop

another state. In Line 11, an edge (i, j), corresponding to (a, b), is added to A_c. In addition, if the i-th segment has another parent j' rather than j, an edge (i, j') is deleted from A_c in Line 13. The parser in Fig. 9 is not incremental, but $|N_c| \leq |N_{t(c)}|$ and $|A_c| \leq |A_{t(c)}|$ hold.

In the remainder of this section, we discuss the time complexity of ParseWithRules. Loops of Lines 3 and 7 in ParseWithRules are repeated $2n - 2$ times, as discussed in Section 2, and $|R|$ times, respectively. If a graph sequence consists of general graphs, the computation time needed to check whether $body(r) \sqsubseteq s_d$ is identical to the subgraph isomorphism is known to be NP-complete [7]. However, since a graph sequence in this paper consists of ordered forests, the computation time to check it is linear with respect to the number of vertices in a forest, which corresponds to the number of segments in a sentence [13]. The complexity of ParseWithRules is therefore $O(n(\theta + |R|n))$. Additionally, in our implementation, for a transformation sequence s'_d of $d\Diamond c$, the computation to check whether $body(r) \sqsubseteq s'_d$ is solvable in constant time by storing mappings between vertices in $body(r)$ and vertices in s_d,; that is, the complexity of ParseWithRules is $O(n(\theta + |R|))$. The complexity of parsing a sentence x is therefore linear with respect to the number of segments n in a sentence, and is equivalent to that of the conventional method.

5 Experiments

We evaluated the proposed method using Kyoto Text Corpus v4.0, which consists of newspaper articles. In the implementation, CaboCha-0.60, which is a representative transition-based parser for Japanese [12], was integrated into the proposed method. We used the period from January 1 to 8 (7635 sentences) to train the SVM. In addition, we used data for eight days between January 9 to 17 (12054 sentences) to mine the rewriting rules and data for one day that is not used to mine the rules in evaluating the proposed method. We repeated this process nine times, which corresponds to nine-fold cross-validation.

We assigned a feature name with value of 1 to each vertex as a vertex label, since a feature vector that characterizes a segment w_i and is processed in the SVM of CaboCha-0.60 is a binary feature vector. In addition, we assigned feature names to each vertex as labels, although the original GTRACE assumes

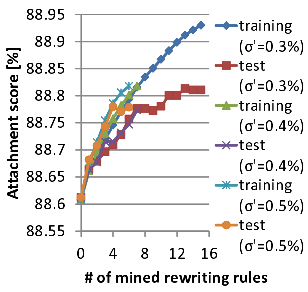

Fig. 11. Attachment score

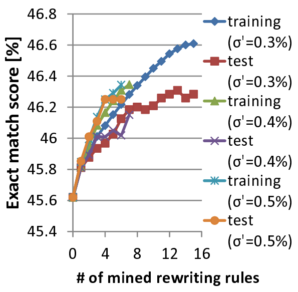

Fig. 12. Exact match score

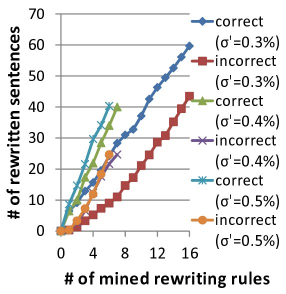

Fig. 13. # of sentences rewritten by rules

that a label is assigned to each vertex in a graph sequence. Therefore, in our experiments, the insertion of a vertex with labels $\{lv_1, \cdots, lv_n\}$ and vertex ID 1 is naturally represented as $\langle tr^{(j,1)}_{[1,lv_1]}, \cdots, tr^{(j,n)}_{[1,lv_n]}\rangle$. For example, the second segment shown in Example 1 is characterized by a set of features $\{$*HON*, *WO*, noun, common_noun, particle$\}$, and a vertex for the segment is labeled by the features.

Figure 10 shows one of the rewriting rules mined using the proposed method under the minimum support threshold 0.5%, where h, i, j, and k are segment IDs satisfying $h < i < j < k$, and the terms in each circle are labels. The rule was mined in the first loop in Line 13 in Fig. 9. The support and confidence of the rule are 0.58% and 89.7%, respectively. Japanese native speakers know that a segment containing *WO* usually modifies the first transitive verb appearing after the segment, and that the segment is the object of the verb in the sentence. However, a segment containing a transitive verb has only one dependent that contains *WO* and appears as the nearest before the verb. In the case that there are two segments containing *WO* before a verb, the former segment modifies a segment containing *NI* appearing after the former segment.

The rewriting rule shown in Fig. 10 mentions that the oracle selects Shift when determining whether the h-th segment is the dependent of the i-th segment in the second state, because a segment containing *WO* usually modifies a transitive verb after the segment, as mentioned above. At this point, the parser does not know that another segment containing *WO* appears between the h-th segment and a segment containing a verb, because the parser is the arc-standard parser. Subsequently, the oracle selects Arc to transit from the fourth state, for the same reason. In the sixth state, the arc-standard parser cannot add an edge (h, i), and it adds an edge (h, k), although the segment containing the verb already has a dependent containing *WO*. This rule rewrites the seventh state by deleting the edge (h, k), and adds an edge (h, i). The rule is therefore valid grammatically.

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.

Figures 11 and 12 show the attachment score and the exact match score of the proposed method for the number of mined rewriting rules for nine-fold cross-validation under the minimum support thresholds 0.3%, 0.4%, and 0.5%[4]. The number of mined rewriting rules $|R|$ increases one by one in each loop in Line 3 of Fig. 9. The number of mined rules is not very large, because some of the rules are unlexicalized and the others are lexicalized not by content words but by functional words (case markers such as WO or NI and auxiliary verbs). The attachment and exact match scores at $|R| = 16$ were improved by 0.20% and 0.66% under the minimum support threshold 0.3%, respectively. The number of mined rules differs for each trial in the nine-fold cross-validation, and the scores of the proposed method were finally improved by 0.22% and 0.90% under the minimum support threshold 0.3%, respectively. Since the number of mined rules is small, the improvements in the scores are not high. However, we can conclude that the mined rewriting rules are valid because improvements in the scores were obtained. In addition, Fig. 13 shows the numbers of sentences rewritten correctly and incorrectly under the various minimum support thresholds for the test datasets. It shows that when the number of rewriting rules mined by the proposed method increases, the numbers of sentences rewritten correctly and incorrectly decrease and increase, respectively, because confidence in the rewriting rules decreases with a progressive increase in the number of rules.

6 Discussion and Conclusion

In this paper, we proposed a method for mining rules for rewriting states reaching incorrect final states, and proposed a dependency parser with rules maintaining time complexity linear with respect to the number of segments in a sentence. The rewriting rules mined by the proposed method are human-readable, and it is possible for us to design new parsers and to generate features in the machine learner in the parser to avoid obtaining incorrect dependency structures. In this paper, we used GTRACE to analyze transition sequences, although there are other data structures for representing graph sequences, such as dynamic graphs and evolving graphs, 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. The class of graph sequences is therefore general enough to apply to the analysis of transition sequences, compared with dynamic graphs and evolving graphs. The principle of the method proposed in this paper can basically be applied to any parsers based on a transition system, including parsers

[4] The attachment and exact match scores of the conventional method trained using data for 15 days were 89.4% and 47.7%, respectively.

employing the beam search [16,8], for sentences in any language. We plan to apply the method proposed in this paper to other transition-based parsers and to corpora of other languages in the future.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proc. of Int'l Conf. on Very Large Data Bases (VLDB), pp. 487–499 (1994)
2. Ahmed, R., Karypis, G.: Algorithms for Mining the Evolution of Conserved Relational States in Dynamic Networks. In: Proc. of IEEE Int'l Conf. on Data Mining (ICDM) (2011)
3. Berlingerio, M., Bonchi, F., Bringmann, B., Gionis, A.: Mining Graph Evolution Rules. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS, vol. 5781, pp. 115–130. Springer, Heidelberg (2009)
4. Borgwardt, K.M., et al.: Pattern Mining in Frequent Dynamic Subgraphs. In: Proc. of IEEE Int'l Conf. on Data Mining (ICDM), pp. 818–822 (2006)
5. Culotta, A., Sorensen, J.: Dependency Tree Kernels for Relation Extraction. In: Proc. of Annual Meeting of Association for Comp. Linguistics (ACL), pp. 423–429 (2004)
6. Ding, Y., Palmer, M.: Automatic Learning of Parallel Dependency Treelet Pairs. In: Su, K.-Y., Tsujii, J., Lee, J.-H., Kwong, O.Y. (eds.) IJCNLP 2004. LNCS (LNAI), vol. 3248, pp. 233–243. Springer, Heidelberg (2005)
7. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York (1979)
8. Huang, L., Sagae, K.: Dynamic Programming for Linear-Time Incremental Parsing. In: Proc. of Annual Meeting of the Association for Computational Linguistics (ACL), pp. 1077–1086 (2010)
9. Inokuchi, A., Washio, T.: A Fast Method to Mine Frequent Subsequences from Graph Sequence Data. In: Proc. of IEEE Int'l Conf. on Data Mining (ICDM), pp. 303–312 (2008)
10. Iwatate, M., et al.: Japanese Dependency Parsing Using a Tournament Model. In: Proc. of Int'l Conf. on Comp. Linguistics (COLING), pp. 361–368 (2008)
11. Kubler, S., et al.: Dependency Parsing. Morgan and Claypool Publishers (2009)
12. Kudo, T., Matsumoto, Y.: Japanese Dependency Analysis using Cascaded Chunking. In: Proc. of Conf. on Comp. Natural Language Learning (CoNLL), pp. 63–69 (2002)
13. Makinen, E.: On the Subtree Isomorphism Problem for Ordered Trees. Information Processing Letters 32(5), 271–273 (1989)
14. Nivre, J.: Algorithms for Deterministic Incremental Dependency Parsing. Comp. Linguistics 34(4), 513–553 (2008)
15. Pei, J., et al.: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In: Proc. of Int'l Conf. on Data Engineering (ICDE), pp. 2–6 (2001)
16. Zhang, Y., Clark, S.: A Tale of Two Parsers: Investigating and Combining Graph-based and Transition-based Dependency Parsing. In: Proc. of Conf. on Empirical Methods in Natural Language Processing (EMNLP), pp. 562–571 (2008)