<table>
<thead>
<tr>
<th>PAPER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency Chart Parsing Algorithm Based on Ternary-Span Combination</td>
</tr>
</tbody>
</table>

**Meixun JIN**, Yong-Hun LEE, Nonmembers, and Jong-Hyeok LEE, Member

**SUMMARY** This paper presents a new span-based dependency chart parsing algorithm that models the relations between the left and right dependents of a head. Such relations cannot be modeled in existing span-based algorithms, despite their popularity in dependency corpora. We address this problem through ternary-span combination during the subtree derivation. By modeling the relations between the left and right dependents of a head, our proposed algorithm provides a better capability of coordination disambiguation when the conjunction is annotated as the head of the left and right conjuncts. This eventually leads to state-of-the-art performance of dependency parsing on the Chinese data of the CoNLL shared task.

**key words:** dependency chart parsing, span-based parsing, factor-based parsing, dependency parsing algorithm, averaged perceptron, syntactic analysis

1. **Introduction**

Chart parsing is frequently used to derive the syntactic structure of a given sentence [1]–[3], and various chart parsing algorithms are available for dependency parsing tree derivation (Sect. 3). Among them, Eisner’s algorithm [4], [5] is notable for its efficiency, with a time complexity of $O(n^3)$. In [2], Eisner’s algorithm has been successfully augmented with current learning methods in the frame of graph-based data-driven dependency parsing. Eisner’s algorithm was further extended in the works of [6], [7] and [3], and all of these parsers reported a state-of-the-art performance on the English dependency treebank that is converted from the English Penn Treebank [8]. We refer to these algorithms as span-based algorithms because they all perform parsing by composing spans (details in Sect. 2).

Our algorithm is another extension of Eisner’s algorithm, and aims to model the relations between the left and right dependents of a head. It is quite common that a head simultaneously dominates the left and right dependents, e.g., the case in which a predicate verb takes subject and object as its left and right dependents in a Subject-Verb-Object (SVO) language, or the case of the coordinate structures in which the conjunction is annotated as the head of the left- and right-side conjuncts. Despite their popularity and the importance of modeling such relations in dependency parsing, they have not yet been modeled in existing span-based dependency parsing algorithms.

In the proposed parsing algorithm, we derive subtrees by augmenting three spans, and the relations between the left and right dependents are modeled during the span-augmentation (Sect. 2.2). Experimental results show that the proposed algorithm provides a better coordination disambiguation when the conjunctions are annotated as heads, and that this improved coordination disambiguation eventually improves the overall parsing performance (Sect. 5).

This paper is organized as follows. We describe the spans and present our algorithm in Sect. 2. In Sect. 3, we compare our algorithm with other dependency chart parsing algorithms. In Sect. 4, we briefly discuss various dependency annotation schemes and their corresponding parsing algorithms. Section 5 presents our experimental results and finally, Sect. 6 provides some concluding remarks regarding our propose.

2. **Proposed Algorithm**

As normal CYK algorithms [9], our algorithm derives a dependency parse tree by recursively combining smaller subtrees in a bottom-up manner. We use a dynamic programming table, known as a chart, to store the partial results for each parsing step.

There are two types of dependency chart parsing algorithms, i.e., constituent-based and span-based algorithms, depending on the type of subtrees that the algorithm processes. If the algorithm constructs subtrees of spans, it is a span-based algorithm, otherwise, it is a constituent-based algorithm. Ours is an improved version of Eisner’s span-based algorithm [4].

2.1 **Span**

A span is a dependency subtree or a sequence of words linked by dependency arcs dominated by the leftmost or rightmost word of the sequence. For example, string $s_{[r,q]}$ shown in Fig. 1 (a) is a span dominated by the leftmost word $s$, which we represent using a right triangle as shown in Fig. 1 (d). The vertical side of the triangle indicates the position of the head. If an internal word $r$ dominates the
string, as shown in Fig. 1 (b), $\text{STRING}_{[q,1]}$ is represented by two spans, as shown in Fig. 1 (e).

Adding a dummy ‘ROOT’ in front of the sentence and allowing the ‘ROOT’ to dominate the sentence, the parse tree of this extended sentence becomes a span. This span is the ultimate one that a span-based parser aims to derive.

2.2 Ternary-Span Combination

A span-based dependency parser derives parse trees by constructing spans in a bottom-up fashion. A larger span is constructed by combining smaller spans. In this subsection, we show how our algorithm combines smaller spans into a larger span.

By adding a new dependency arc to connect $s$ and $r$, we can combine $\text{STRING}_{[x:l]}$ (Fig. 1 (a)) and $\text{STRING}_{[q+1:r]}$ (Fig. 1 (b)) into $\text{STRING}_{[x:r]}$ (Fig. 1 (c)). $\text{STRING}_{[x:l]}$ corresponds to $\text{SPAN}_{[s]}$ in Fig. 1 (d), while $\text{STRING}_{[q+1:r]}$ corresponds to two spans, i.e., $\text{SPAN}_{[q+1:t]}$ and $\text{SPAN}_{[t]}$ shown in Fig. 1 (e). Thus, this derivation of $\text{STRING}_{[x:r]}$ can be represented by combining three spans into one, as shown in Figs. 1 (d)–(f). We call it ternary-span combination. In a similar way, the right-side head span construction is represented as shown in Figs. 1 (g)–(i).

The pseudocode for ternary-span combination is given in Fig. 2 and explained in Sect. 2.3.

Figures 1 (a)–(c) show one derivation of $\text{STRING}_{[x:l]}$. To derive the parse tree of a string we must select a pair of substrings that lead to an optimal parse tree. For $\text{STRING}_{[x:l]}$ in Fig. 1 (c), we may select the optimal pair of strings, i.e., $\text{STRING}_{[x:q]}$ and $\text{STRING}_{[q+1:t]}$, by enumerating all possible values of $q$ and $r$ (Lines 11 and 16 of the pseudocode in Fig. 3). Here, $q$ is the boundary of the two strings concatenated into $\text{STRING}_{[x:l]}$, and $r$ is the newly selected dependent of $s$ during the parse step.

2.3 The Algorithm

As a normal chart parser, we store the max score of a span and its related parameters, i.e., $q$ and $r$, for each parse step in the corresponding cells of the chart. For convenience, we use $K_{[x:l]}$, $B_{[x:l]}$, and $D_{[x:l]}$ to represent the indices of two parameters $q$ and $r$ that give the maximum score $K_{[x:l]}$.

The pseudocodes used to calculate $K_{[x:l]}$, $B_{[x:l]}$, and $D_{[x:l]}$ are given in Fig. 3. Line 11 of Fig. 3 shows that, to find the maximum score $K_{[x:l][‘L’]}$ of $\text{SPAN}_{[x:l][‘L’]}$, we need to find the indices $q$ and $r$ that lead to the maximum score through $\text{TernaryCombination}$ (the pseudocode in Fig. 2). After enumerating all possible values of $q$ and $r$, Line 11 selects the ternary-span combination that gives the highest score. The maximum score is then assigned to $K_{[x:l][‘L’]}$ (Line 12), and the $q$ and $r$ ($Q,R$, Line 11) that give the maximum score are assigned to $B_{[x:l][‘L’]}$ and $D_{[x:l][‘L’]}$, respectively (Lines 13 and 14). In a similar way for the right-side head $\text{SPAN}_{[x:l][‘R’]}$, the related derivations for $K_{[x:l][‘R’]}$, $B_{[x:l][‘R’]}$, and $D_{[x:l][‘R’]}$ are given in Lines 16–19.

The module $\text{TernaryCombination}$ (Fig. 2) calculates the score of the span generated by combining three smaller spans. Deriving the boundary indices of the three spans ($s,q,+l:r$), from its input parameters (Line 1 of Fig. 2), the score of the newly generated span is the accumulated scores of the three spans plus the score of the new dependency arc linking them together. In Line 3, $K_{[x:l][‘L’]_t}$, $K_{[q+1:r][‘R’]_t}$, and $K_{[r][‘R’]}$ are the scores of the three spans combined into $\text{SPAN}_{[x:l][‘L’]}$, and the $\text{dep}(\ast)$ function evaluates the dependency arc linking them. Similarly, Line 5 calculates the score for the right-side head $\text{SPAN}_{[x:l][‘R’]}$.

The function $\text{dep}(\ast)$ evaluates the new dependency arc. In addition to the indices of the target head $s$ and dependent $r$ (Line 3 in Fig. 2), $\text{dep}(\ast)$ uses the parse history (parse decisions from the previous parse steps), i.e., $D_{[x:l][‘L’]}$, $D_{[q+1:r][‘R’]}$, and $D_{[r][‘R’]}$. This parse history is closely related to the three spans in processing: $D_{[x:l][‘L’]}$ is the dependent of $s$ selected for the construction of $\text{SPAN}_{[x:l][‘L’]}$, while $D_{[q+1:r][‘R’]}$ and $D_{[r][‘R’]}$ are the dependents of $r$, selected for $\text{SPAN}_{[q+1:r][‘R’]}$ and $\text{SPAN}_{[r][‘R’]}$, respectively.

Here, we include the parse history in $\text{dep}(\ast)$ to model the relation between the left and right dependents of $r$. By including the left-side dependent $D_{[q+1:r][‘R’]}$ and right-side dependent $D_{[r][‘R’]}$ of $r$ in $\text{dep}(\ast)$ at Line 3 of Fig. 2, we evaluate the relations among $(r,D_{[q+1:r][‘R’]},D_{[r][‘R’]})$, simultaneously, along with the evaluation of the dependency arc connecting $s$ and $r$. Regarded as out-of-span relations [10], the relations between the left and right dependents have been ignored in existing span-based algorithms, and to the best of our knowledge, this research may be the first attempt to model them in a span-based approach.

The parse tree with the maximum score, which is derived for the input sentence, is given by $\text{SPAN}_{[x:l][‘L’]}$ for a sentence length of $n$, and the parse tree can be constructed by backtracking the parameters recorded in $B_{[x:l]}$ and $D_{[x:l]}$.

With a simple look through the pseudocode (Figs. 2 and 3), the overall time and space complexities of the algorithm are $O(n^4)$ and $O(n^3)$, respectively.
3. Comparison with Other Algorithms

Considering certain parse history is an attempt to capture syntactic relations beyond the explicitly-represented by dependency arcs. There has been a recent trend of considering more parse history in graph-based approaches ranging from one-side sibling [6] to tri-sibling [3]. Table 1 lists seven different dependency chart parsing algorithms. These algorithms differ in the method used for building larger subtrees from smaller subtrees. In this section, we focus on how these algorithms differ in using parse history, when determining a new dependency arc connecting from \( h \) (head) to \( d \) (dependent).

3.1 Constituent-Based Algorithms

Algs. 1–5 in Table 1 are constituent-based algorithms. Table 1 shows the step(s) these algorithms use to derive subtree \( \mathcal{L} \) by combining the two subtrees \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \). Here, \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) represent the subtrees dominated by \( h \) and \( d \), respectively; \( \mathcal{L}_3 \) represents the newly constructed subtrees produced by combining \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \).

The derivation of \( \mathcal{L}_3 \) in Alg. 1 (Table 1) is processed in a single step by augmenting \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) on the basis of their dependency. The notation of \( K(\mathcal{L}_3) \) in Table 1, represents the score of subtree \( \mathcal{L}_3 \). In Alg. 1, this is calculated as: \( K(\mathcal{L}_3) = K(\mathcal{L}_1) + K(\mathcal{L}_2) + \text{dep}(h, d) \). We define the dependency function \( \text{dep}(\mathcal{L}_3) \) as: \( \text{dep}(h, d, \mathcal{L}_1, \mathcal{L}_2) \), which indicates that the available parse history for Alg. 1 includes \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) (represented by the dashed arcs in Fig. 4 (a)).

Algs. 2–5 are variants of Alg. 1, and are based on two-step operations. Two-step processing improves the computational complexity of Alg. 1 from \( O(n^5) \) to \( O(n^4) \).

In Alg. 2, the first step generates a partial result \( \mathcal{L}_4 \), and evaluates the possibility that word \( h \) dominates \( \mathcal{L}_4 \). Score \( K(\mathcal{L}_4) \) is defined as \( \text{dep}(\mathcal{L}_3) + K(\mathcal{L}_4) \). The second step combines \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) into a new constituent. Evaluating the dependency between \( h \) and \( d \) in the first step, the available parse history for Alg. 2 is \( (h, \mathcal{L}_1) \), which excludes all the selected dependents of \( h \) (Fig. 4 (b)). Alg. 3 is similar to Alg. 2. Its available parse history is \( (\mathcal{L}_1, d) \), which excludes all the dependents of \( h \) (Fig. 4 (c)).

In Algs. 4 and 5, one of the constituents is divided into two parts (spans). The first step combines the constituent with the closest span by evaluating the dependency between them. During the second step, the other span is attached to the result from the first step to form a new constituent. Now, the parse history is \( (\mathcal{L}_1, \mathcal{L}_2) \) for Alg. 4 (Fig. 4 (d)), and \( (\mathcal{L}_1, \mathcal{L}_2) \),
Table 1  Overview of dependency chart parsing algorithms: Algs. 1–5 are constituent-based and Algs. 6–7 are span-based.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>First step</th>
<th>Second step</th>
<th>Score of the new subtree</th>
<th>Dependency evaluation function dep(•)</th>
<th>Time/Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg. 1</td>
<td><img src="image1" alt="Alg. 1" /></td>
<td><img src="image2" alt="Alg. 1" /></td>
<td>$K (\xrightarrow{h,d,r}) = K(\xrightarrow{h,r}) + K(\xrightarrow{d,r}) + \text{dep}(•)$</td>
<td>dep(•) = dep(h, d, (\xrightarrow{h,d,r}))</td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
<tr>
<td>Alg. 2</td>
<td><img src="image3" alt="Alg. 2" /></td>
<td><img src="image4" alt="Alg. 2" /></td>
<td>$K (\xrightarrow{h,d,r}) = \text{dep}(•) + K(\xrightarrow{h,d,r})$</td>
<td></td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
<tr>
<td>Alg. 3</td>
<td><img src="image5" alt="Alg. 3" /></td>
<td><img src="image6" alt="Alg. 3" /></td>
<td>$K (\xrightarrow{h,d,r}) = K(\xrightarrow{h,d}) + K(\xrightarrow{d,r})$</td>
<td>dep(•) = dep(h, d, (\xrightarrow{h,d,r}))</td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
<tr>
<td>Alg. 4</td>
<td><img src="image7" alt="Alg. 4" /></td>
<td><img src="image8" alt="Alg. 4" /></td>
<td>$K (\xrightarrow{h,d,r}) = K(\xrightarrow{h,d}) + K(\xrightarrow{d,r}) + \text{dep}(•)$</td>
<td></td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
<tr>
<td>Alg. 5</td>
<td><img src="image9" alt="Alg. 5" /></td>
<td><img src="image10" alt="Alg. 5" /></td>
<td>$K (\xrightarrow{h,d,r}) = K(\xrightarrow{h,d,r}) + \text{dep}(•)$</td>
<td></td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
<tr>
<td>Alg. 6</td>
<td><img src="image11" alt="Alg. 6" /></td>
<td><img src="image12" alt="Alg. 6" /></td>
<td>$K (\xrightarrow{h,d,r}) = K(\xrightarrow{h,d,r}) + \text{dep}(•)$</td>
<td>dep(•) = dep(h, d, (\xrightarrow{h,d,r}))</td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
<tr>
<td>Alg. 7</td>
<td><img src="image13" alt="Alg. 7" /></td>
<td><img src="image14" alt="Alg. 7" /></td>
<td>$K (\xrightarrow{h,d,r}) = K(\xrightarrow{h,d,r}) + \text{dep}(•)$</td>
<td></td>
<td>$O(n^5)/O(n^3)$</td>
</tr>
</tbody>
</table>

Fig. 4  Available parsing history (represented as dashed lines), for each parsing algorithm when determining dependency between h and d.

4 For more information on Algs. 2 and 4, refer to [5]. Algs. 3 and 5 can be implemented in a similar way.

3.2 Span-Based Algorithms

In Table 1, Algs. 6–7 are span-based algorithms; they derive a new span $\xrightarrow{h\,d\,r}$ by combining $\xrightarrow{h\,d}$ and $\xrightarrow{r}$ and evaluating the dependency between them. In the second step, $\xrightarrow{h\,d\,r}$ is extended to $\xrightarrow{h\,d\,r}$ by combining with $\xrightarrow{d\,r}$. This algorithm is notable for its efficiency, with a computational complexity of $O(n^3)$. The parse history available for Alg. 6 include two spans $\xrightarrow{h\,d}$ and $\xrightarrow{r}$, as shown in Fig. 4 (f).

Alg. 7 is the proposed algorithm, with a parse history of $\xrightarrow{h\,d\,r\,a\,d\,r}$, as shown in Fig. 4 (d).
Table 2 The subtrees modeled by each span-based algorithm. The dashed arcs are the parse history used for determining the solid arcs.

<table>
<thead>
<tr>
<th>Feature Types</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAt.ARC</td>
<td>uni-gram, bi-gram, surrounding in-between</td>
</tr>
<tr>
<td>FEAt.ONE-SIDE.SIB</td>
<td>h-POS, d-POS sib-POS, d-POS sib-POS</td>
</tr>
<tr>
<td>FEAt.GRAND.CHILD1</td>
<td>h-POS, d-POS right-POS, d-POS right-POS</td>
</tr>
<tr>
<td>FEAt.GRAND.CHILD2</td>
<td>h-POS, d-POS left-POS, d-POS left-POS</td>
</tr>
<tr>
<td>FEAt.TWO-SIDE.SIB</td>
<td>h-POS, d-POS left-POS right-POS, d-POS left-POS right-POS</td>
</tr>
</tbody>
</table>

3.3 Comparison

Alg. 6 in Table 1 uses the least amount of parse history, yet, it is the most efficient; Alg. 1 uses the greatest amount of parse history, but has the highest complexity. Of the algorithms with a time complexity of \(O(n^2)\), Algs. 4 and 5 use more parse history than Algs. 2 and 3. Our algorithm uses a similar parse history as Alg. 4 (Fig. 4 (d)), but ours obtains a better space complexity of \(O(n^2)\), compared with \(O(n^3)\) for Alg. 4.

The parse history used by each span-based algorithm is clearly specified in Table 2. The 3rd-order algorithm models more relations than the 1st-, 2nd- and high-order algorithms (Table 2). In addition to one-side sibling (Table 2 (b)) and grandparent relations (Table 2 (c)), the 3rd-order algorithm models tri-sibling and grand-sibling relations as shown in Tables 2 (d) and (e).

As shown in Fig. 5 (a), our algorithm can model parse history involved in three spans. Figures 5 (b)–(e) show some of the relations enclosed in Fig. 5 (a), which are modeled in our algorithm (Table 4). The prominent difference between our algorithm and other span-based algorithms is that existing span-based algorithms focus on modeling, or are biased in modeling dependency arcs that point toward the same direction, i.e., either from left to right (all dependency arcs in

Tables 2 (a)–(e) are in this direction), or from right to left. Such a bias is more significant for the third-order algorithm by modeling the relations of tri- and grand-siblings. The proposed algorithm is helpful to remove this bias by modeling two-side siblings (dashed arcs in Table 2 (f)).

4. Dependency Annotation and Parsing Algorithm

Previous work has shown that the use of parse history is crucial to achieve a high performance parsing. The comparison provided in Sect. 3 shows the trade-off between a decrease
in parse complexity and utilizing a greater amount of parse history. Thus, we must choose a parsing algorithm that balances the parse complexity and the use of the parse history.

Another factor related to the appropriate selection of a parse algorithm is the structural bias of the corpus. According to the description given in [11], the structural bias of a corpus is the tendency for a corpus to contain more specific types of structured subtrees. For example, the English dependency treebank (abbreviated as English data) converted from the English Penn Treebank [8] using Penn2Malt† contains more of the subtree as Table 3 (a), compared with the Chinese treebank of CoNLL’07 (abbreviated as Chinese data).

In addition to the inherent characteristics of the language, structural bias of a corpus is partly due to the adopted dependency annotation method. Depending on the dependency annotation scheme, a syntactic relation may be represented differently. For example, we can annotate the Coordinate Conjunction as the Head (CCH-type), with the left and right conjuncts as its dependents; whereas other methods may set the Coordinate Conjunction as the Dependent (CCD-type) by annotating one of the conjuncts as its head [10].

As SVO languages, both English and Chinese share many syntactical similarities. Yet, the portion of the heads that dominating triple one-side dependents (tri-siblings as subtree of Table 3 (a)) in the English data is 3.73%, which is much higher than that in the Chinese data. The reason for this is partly due to the difference in annotating coordinations. In the English corpus, most verbal-coordinations are dominated by their left-most conjunct†† [12], upon which other conjuncts and conjunction(s) are attached. Thus, it is necessary to model the subtree of Table 3 (a) in English corpus for coordination disambiguation. In this sense, among the algorithms of Table 2, the 3rd-order algorithm [3] is the better choice for the English corpus. Actually, the system of [3] includes some features, with which the 3rd-order algorithm gives better coordination disambiguations.

The Chinese corpus is one with CCH-type coordinations. Considering the importance for coordination disambiguation in parsing, an algorithm that models the subtrees shown as Table 2 (f) (the pair of arcs shown with the dashed lines) is a better choice.

5. Experiments

We evaluated the performance based on an Unlabeled Attachment Score (UAS), following the measure defined in the CoNLL shared task [13], [14]. Most data-driven dependency parsers [6], [15] adopt two-stage for labeled parsing by deriving dependency labels at the second-stage upon the unlabeled dependency tree of the first-stage. Multi-labeled classifiers are often adopted to label each dependency arc. In this sense, the proposed algorithm has little effect on dependency labelling, we did not evaluate with Labeled Attachment Score (LAS) in this paper.

The parsing model in this paper is conventionally defined by selecting the maximum score parse tree. The score of the parse tree is the sum of the scores of the dependency arcs. The dependency function dep(*) (included in Lines 3 and 5 of Fig. 2) is defined as:

$$\text{dep}(*) = \text{Feature}(*) \times \text{Weight}(*)$$

Feature(*) represents the features related to the target dependency arc, while Weight(*) is the associated weight function, which is learned through the average perceptron [16]. The average perceptron allows for fast learning with a respectable performance and a simple implementation, compared with other structured training algorithms, such as the max-margin model [17], the margin infused relaxed algorithm (MIRA) [6] and the log-linear model [15].

5.1 Features

Our algorithm can model the relations represented by the subtrees shown in Fig. 5; we define the five types of features accordingly, as shown in Table 4. The Fea.arc features are directly related to the target dependency arc, including the uni-gram, bi-gram, surrounding and in-between features, as defined in [18]. Other types of features are targeted to model the one-side sibling, grand-child and two-side sibling relations, respectively.

In Table 4, for example, we use Fea.oneside.sib to represent features modeling the one-side sibling relation between the arc linking from h to d and arc connecting from h to sib (Fig. 5 (b)). Since the pair of arcs in one-side sibling indicates the relation among three nodes: h, d, and sib, we define Fea.oneside.sib to include a POS tag tri-gram of h, d and sib (represented as h-POS d-POS sib-POS in Table 4), and a POS tag bi-gram of d and sib (d-POS sib-POS). The other feature types in Table 4, are defined in a similar way.

Similar features of Fea.oneside.sib, Fea.grandchild1 and Fea.grandchild2 has been defined and used in the systems of [6], [18], and [7], while, Fea.twoside.sib features in Table 4, are newly defined for our algorithm. The Fea.twoside.sib features are defined primarily to evaluate the relations between the left and right dependents of d shown as Fig. 5 (e).

5.2 Results

We evaluated our system on Chinese CoNLL-2007 data and the English data converted from the English Penn Treebank. For the Chinese data, we used the original included training and test sets; for the English data, we used the sections 2–21 for training and the section 23 for test as convention. For both data sets, we used the gold standard part-of-speech tags in the treebanks.

†Penn2Malt is an automatic tool for converting a Penn Treebank style phrase structure tree into dependency tree. http://w3.msi.vxu.se/nivre/research/Penn2Malt.html

††In case of a nominal-coordination, the right-most conjunct is annotated as the head in the English data with Penn2Malt.
5.2.1 Effect of Each Feature Type

First, we evaluated the performance of our algorithm with each type of features described in Table 4. As shown in Table 5, adding features modeling parse history such as Fea. oneSide.Sib, Fea. grandChild1, Fea. grandChild2 or Fea. twoSide.Sib on base feature Fea. arc, results in better performances than using only Fea. arc type features.

With Fea. oneSide.Sib, our system obtained performances of 91.20% and 88.40% for the English and Chinese data, respectively; these performances are better than those obtained from Fea. grandChild1 and Fea. grandChild2 (Table 5). These results also indicate that Fea. oneSide.Sib is one of the most important features for both data.

With Fea. twoSide.Sib, our system obtained a performance of 88.58% for the Chinese data, and this performance is better than the performance with other types of features.

5.2.2 Effect of Modeling Two-Side Siblings

CCH-type coordinate are the primary case that require to model two-side siblings. Table 6 shows that with Fea. twoSide.Sib, 83.78% of the conj. head†† are correctly attached to their heads in the Chinese data, which is an improvement of 1.48%, compared to the result using Fea. oneSide.Sib.

Including a coordination-specialized feature Coor.Fea, defined as:

```
if left_conj.head.POS == right_conj.head.POS  coor.Fea = 1
else  coor.Fea = 0
```

the system provides more accurate results on coordination disambiguation than using only Fea. twoSide.Sib (Table 6). Furtherly, as shown in Table 7, Coor.Fea is more helpful for identifying remote conj. heads from their conjunctions. Here, Coor.Fea is a flag-like feature that provides symmetric information for the coordination by comparing the POS tags of its conjuncts.

The case of a verb dominating a subject at its left-side and an object at its right-side (sub_VERB_obj), is a typical syntactic structure for a SVO language. With Fea. twoSide.Sib, about 89.72% of sub_VERB_objs are correctly identified for the English data, which was 7.70% higher than the result of 82.02% gained with Fea. oneSide.Sib.

However, the portion of sub_VERB_obj is rather lower than expected in the English data. Among test sentences (totally 2,416), only 336 sentences are dominated by sub_VERB_obj type ROOTs. The reason for such a low portion is partly due to the existence of so many sub_VERB_vmod ROOTs (total 1,146 cases) and annotation scheme assigning auxiliary verb as the head of main verb (614 cases) in the English data. For cases of sub_VERB_vmod and sub_VERB_vc (the case including auxiliary verb), the effect of modeling two-side sibling is limited since the correlations between left- and right-side dependents of Verb in these cases are weak. And because of the low portion of sub_VERB_obj, modeling two-side siblings has little effect on the overall performance in the English data.

5.2.3 Effect of Combined Features

We then evaluated our system empirically on the combined features. For the Chinese data, the best result was 89.41% (Table 8) with (Fea. oneSide.Sib + Fea. twoSide.Sib + Coor.Fea).

For the English data, the best performance of our system was 91.99%, which was obtained using (Fea. oneSide.Sib + Fea. grandChild1 + Fea. grandChild2) (Table 8).

<table>
<thead>
<tr>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>Fea. twoSide.Sib</td>
</tr>
<tr>
<td>Fea. twoSide.Sib + Coor.Fea</td>
</tr>
</tbody>
</table>

*The distance is the number of words that appeared between the conj.head and conjunction. For example, given 'Tom and the pretty Jerry', Tom and Jerry are 0-distance and 2-distance conj.heads, respectively.

Table 8 Performances of the proposed algorithm on combined features.

<table>
<thead>
<tr>
<th>Name</th>
<th>Features</th>
<th>UAS (English)</th>
<th>UAS (Chinese)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed.I</td>
<td>Fea. oneSide.Sib + Fea. grandChild1 + Fea. grandChild2</td>
<td>91.99%</td>
<td>88.49%</td>
</tr>
<tr>
<td>Proposed.II</td>
<td>Fea. oneSide.Sib + Fea. twoSide.Sib + Coor.Fea</td>
<td>91.03%</td>
<td>89.41%</td>
</tr>
</tbody>
</table>

††Fea.arc features were included in each experiment.
††† conj.head refers to the head word of a conjunct. In CCH-type coordination, each conj.head is attach to the conjunction.
5.3 Comparison with Other Parsers

5.3.1 Performance

Among the systems listed in Table 9, the 2nd-order system and ours, proposed.I and proposed.II, are based on POS tagged features. The 2nd-order system was based on \texttt{FEA:ONE-SIDE:SUB}. In addition to \texttt{FEA:ONE-SIDE:SUB}, proposed.I and proposed.II model more types of features (Table 8). For the English data, proposed.I obtained a performance of 91.99%, which outperformed the 2nd-order system (Table 9). For the Chinese data, proposed.II gave the best performance by including \texttt{COORD:FEA} features.

In Table 9, the systems of [3], [7], and [19] include surface words in their feature vector, and proposed.I-word is the version using word features based on the same feature template defined for proposed.I. Proposed.I-word gave a performance of 92.39% for the English data, which is better than the performance of [19], and lower than the performance obtained by the 3rd-order system of [3]. In our opinion the performance gap between the 3rd-order system and ours is on modeling tri-siblings, especially, disambiguating CCD-type of coordination (e.g., Table 3 (a)) in the 3rd-order system. We compared the performance of our system on the Chinese data with two top-ranked systems [15] and [20] for the CoNLL’07 shared task, since the systems of [3], [19] did not evaluated on the same Chinese data. Comparing the performances of [15] and [20], our systems gave the best performances on the Chinese data.

The work of [3] also reported the performance on the development data set (the section 24 of the English Penn treebank) as 93.49%. According to report of [3], the emulated high-order system of [7] gave a similar (or slightly lower) performance as 93.14% on the same data set. Despite of the high performance for the English data, the high-order system gave a performance of 86.20% for the Chinese data (Table 9). In our opinion, lack of modeling two-side siblings is the main reason for the low performance of [7] on the Chinese data.

5.3.2 Efficiency

We implemented our systems in Python and ran on a single machine of 64-bit Intel i5 quad-core with 3.3 GHz CPUs. The proposed.I-word system spent totally 41.2 hours on training (for 10 iterations).

Both the 3rd-order system and our systems have a time complexity of $O(n^3)$, while, our algorithm is more efficient. When comparing the the total time spent for decoding the English test data, proposed.I-word spent 489 seconds, while the emulated 3rd-order system spent much more time than ours (Table 10).

6. Conclusion

This paper proposed a new span-based dependency chart parsing algorithm, that is suitable for a language or corpus where modeling the left and right dependents is essential to achieve a high parsing performance.

By modeling the relation between the left and right dependents of a head, the algorithm provides a solution for coordination disambiguation when the conjunction is annotated as the head of its left and right conjuncts. When applied to the Chinese data of the CoNLL’07 shared task, our algorithm achieved a better parse performance on the coordinate structures, which eventually improved the overall parsing performance.

We believe our algorithm can also achieve better performances for Arabic, Czech, and Slovene corpora of CoNLL 2007 [10] than existing span-based algorithms through an improved coordination disambiguation, since the coordinations in these corpora are also of CCH-type. In case of SVO languages, the proposed algorithm also provides a platform to model the relation between the subject and object with regard to their common predicate verb.

Acknowledgments

This work was supported in part by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korean government (MEST No. 2012-0004981), in part by the BK 21 Project in 2012, and in part by IT Consilience Creative Program of MKE and NIPA (C1515-1121-0003).

References


