Abstract

To resolve structural ambiguities in syntactic analysis of natural language, which are caused by prepositional phrase attachment, relative clause attachment, and so on, we developed an experimental system called the Dependency Analyzer. The system uses instances of dependency structures extracted from a terminology dictionary as a knowledge base. Structural (attachment) ambiguity is represented by showing that a word has several words as candidate modifiees. The system resolves such ambiguity as follows. First, it searches the knowledge base for modification relationships (dependencies) between the word and each of its possible modifiees, then assigns an order of preference to these relationships, and finally selects the most preferable dependency. The knowledge base can be constructed semi-automatically, since the source of knowledge exists in the form of texts, and these sentences can be analyzed by the parser and transformed into dependency structures by the system. We are realizing knowledge bootstrapping by adding the outputs of the system to its knowledge base.

1 Introduction

The bottleneck of sentence analysis, structural ambiguity, occurs when a sentence has several alternatives for modifier-modifiee relationships (dependencies) between words or phrases. This kind of ambiguity cannot be resolved merely by applying grammatical knowledge: there is a need for semantic processing. Resolution of structural ambiguities seems to be a problem of selecting the most preferable dependency from several candidates by using large-scale knowledge on dependencies among words. There are two problems in realizing practical semantic processing: one is that knowledge must be large-scale, and must be constructed automatically or semi-automatically; the other is that the mechanism for utilizing knowledge, inference, must be efficient or tractable. We developed a system called the Dependency Analyzer that resolves these problems.

The Dependency Analyzer is a system for structural disambiguation. One of its characteristics is that it selects the most preferable dependency by using a knowledge base containing terminological knowledge in the form of dependency trees. The knowledge base can be constructed semi-automatically, as described in Section 2. The inputs of this system are parse trees, which are outputs of the PEG parser, a broad coverage English parser [5]. The system translates the phrase structures into dependency structures that explicitly represent modifier-modifiee relationships between words. The main processes of the system are executed if attachment ambiguities are included in the phrase structures. In the dependency structures, attachment ambiguities are represented by showing that some words have several candidate modifiees. From these dependency structures, several candidate dependencies are extracted. The system decides which of these should be adopted by using background knowledge and context. The decision is made via the mechanisms of path search and distance calculation. A precise description of path search is given in Section 3. An explanation of distance calculation is given in Section 4. Another problem for disambiguation, namely interaction (or constraints) between attachment ambiguities, is discussed in Section 5.

2 Knowledge Base

The knowledge must be large-scale, since natural language semantics should have a broad coverage of lexical items. Since dependency structures are built by analyzing sentences and by transforming phrase structures in a straightforward way, if knowledge is assumed to consist of dependency structures, a knowledge base is easily constructed by using already-existing on-line dictionaries. This idea of using on-line dictionary definitions as a knowledge base was originally proposed by Karen Jensen and Jean-Louis Binot [6]. Jun-ichi Nakamura and Makoto Nagao [10] evaluated the automatic extraction of semantic relationships between words from the on-line dictionary. We emphasize that a data structure for representing knowledge should be as simple as possible, because it must be easy to construct and efficient.

We selected the tree structure as a means of representing knowledge, because it is a very simple and manageable data structure, and because tree structures are suitable for describing dependency structures. The tree structure is defined as follows. A Tree consists of a Node and recursions (or null) of Tree, and a Node consists of repetitions of a paired attribute name and attribute value.

The tree structure defined in Figure 1 shows a tree (dependency) structure for the clause “the operating system stores the files in the disk.” In this tree, “WORD,” “POS (part of speech),” and “CASE” are attribute names, and “store,” “VERB,” and “AGENT” are attribute values.

In our system, the knowledge can be extracted from dictionaries of terminology, and is of two types: (1) dependency structures and (2) synonym and taxonym relation-
The process of knowledge extraction is as follows. First, dictionary statements are rewritten manually as simple sentences. Next, sentences are parsed into phrase structures by the PEG parser. Then, phrase structures are transformed into dependency structures by the Dependency Structure Builder, which is a component of the Dependency Analyzer. Finally, semantic case markers are manually added to the modification links in dependency structures. Synonym and taxonomy relationships are extracted from sentences of the form “X is a synonym for Y” and “X is a Y” respectively. These sentences are automatically transformed into tree structures each of which has two nodes for the words “X” and “Y” and a link from “X” to “Y” with the label “isa.” In the case of “X is a synonym for Y,” since “Y” is also a synonym for “X,” “Y” is connected with “X” at the same time by a link with the label “isa.” We developed an interactive tree management tool, the Tree Editor, which makes it easy for users to deal with trees.

Another problem of natural language processing is the knowledge acquisition bottleneck. Some ideas on how to acquire knowledge from already-existing dictionaries automatically or semi-automatically have been proposed [10,41. But it is still difficult to develop a knowledge base fully automatically because of ambiguities in the natural language analysis of dictionary definitions. A more practical way to overcome the bottleneck is so-called knowledge bootstrapping. By knowledge bootstrapping, the Dependency Analyzer extends its knowledge automatically by using a core knowledge base that includes manually edited dependency structures. Since the Dependency Analyzer uses dependency structures as knowledge and outputs a dependency structure with no ambiguity (case ambiguity is also resolved by the system), the output can be added to the knowledge base. Of course we still need to evaluate the automatically constructed knowledge base. But the reliability (performance) of the knowledge base is rising gradually, so it is expected that human intervention will be greatly reduced in the near future.

3 Path Search - An Efficient Algorithm

Path search is a process for finding relationships between the words in a candidate dependency by using a knowledge base. Since relationships between words in these candidates do not always exist in the knowledge base, relationships between synonyms and taxonyms of these words can also be targets. Path search is done in the following steps:

1. Synonyms and taxonyms of words in the candidate dependencies are found by using the knowledge base. In the knowledge base, synonym and taxonym relationships are also defined in the form of trees. All the synonyms and taxonyms can be collected by transiting relationships.

2. Dependencies between elements of each synonym and taxonym set (including the original words) are also found by using the knowledge base.

We developed an efficient algorithm for path search, using the table of indices shown in Table 1. In this table, tσ represents the pointer of the tree in which the word on the same line appears, and the numbers in parentheses represent the node location of the word in the tree. The relationships between the numbers and the node are shown in Figure 2. The left side of the table shows trees in which a synonym or a taxonym of the word on the same line appears as its parent node. For example, in the tree to, the word a is on the node of location (0), and by traversing tσ up by one node from location (0) we can find that the word b is on the node of location (0), so b is a synonym or a taxonym of a as shown in Figure 3. Thus, in order to find a synonym or a taxonym of a word, we just traverse up the tree on the left side of the table by one node. We assume that synonym and taxonym relationships are transitive, that is, that a synonym/taxonym of one of the synonyms/taxonyms of a word is also a synonym/taxonym of the word itself. We can

Table 1: Tree Index Table

| word | synonym and taxonomy trees | dependency trees |
|------|---------------------------|------------------|
| a    | tσ(0) tσ(0) tσ(0)       | tσ(0) tσ(0) tσ(0) |
| b    | tσ(0) tσ(0) tσ(0)       | tσ(0) tσ(0) tσ(0) |
| c    | tσ(0) tσ(0) tσ(0)       | tσ(0) tσ(0) tσ(0) |
| d    | tσ(0) tσ(0) tσ(0)       | tσ(0) tσ(0) tσ(0) |

Figure 2: Tree and Node Location

Figure 3: Synonym/Taxonym Tree
collect all its synonyms/taxonyms by iteration of that process. The next stage of path search is to find whether there are dependencies between words within each set of synonyms/taxonyms. This process searches trees that involve both words and checks whether there is a path from one word to the other. In the dependency trees, the words’ locations show whether there is a dependency between them.

For example, we can see that the word $b$ is a dominator of the word $d$ from the locations of these words in the common tree $t_{110}$ (shown in Figure 4), which is included in both the set of dependency trees that include $b$, $\{t_{110}, t_{110}\}$, and that of dependency trees that include $d$, $\{t_{15}, t_{110}\}$. In the tree structures, if the node $a$ is an ancestor of the node $b$, then there is a unique path from $b$ to $a$. Thus, finding dependency between words is equivalent to checking their node locations in the dependency trees. A path between words $w_1$ and $w_2$ is found by the following processes:

1. The synonym/taxonym sets of these words, $S_{w_1}$ and $S_{w_2}$, are collected.

2. The common trees $t_x \ldots$ that involve both elements, $e_i \in S_{w_1}$ and $e_j \in S_{w_2}$, of each set are found.

3. The node locations of $e_i$ and $e_j$ in $t_x \ldots$ are checked.

For example, a path between the words $a$ and $c$ is shown in Figure 5.

4 Distance Calculation - A Heuristic Process for Selection of the Most Preferable Dependency

Several conditions are added to paths, and the closeness of dependency in a path is computed according to these conditions. The degree of closeness of dependency is called the dependency distance. This is calculated by using the number of dependencies included in a path and the values of the conditions. Three conditions are used to calculate the dependency distance:

1. Case consistency

   For example, in the sentence “VM/SP keeps the information on the virtual disk,” there is a prepositional phrase attachment ambiguity, as shown in Figure 6. If the path shown in Figure 8 is found together with the candidate dependency shown in Figure 7, then the semantic case of the path’s dependency between “store” and “disk” must be consistent with the grammatical case of the sentence’s dependency between “keep” and “virtual disk.” Here, the case consistency between the sentence and the path holds, since the grammatical case “on” can have the role of the semantic case “location.” If this consistency holds, then the value of case consistency is 1; otherwise, it is 0.

2. Co-occurrence consistency

   This is the consistency between the other modifiers of the modifiee of the candidate dependency, called the co-occurrence modifiers, and those of a path.

For example, a path between the words $a$ and $c$ is shown in Figure 8.
3. Co-occurrence consistency

In the example sentence, for instance, there is a co-occurrence modifier "VM/SP" of the candidate dependency between "keep" and "virtual disk," as shown in Figure 9. In this case, "VM/SP" has the grammatical case subject. On the other hand, if the path is given by the dependency tree shown in Figure 10, then there is also a co-occurrence modifier "operating system" that has the semantic case of agent. In addition, there is a taxonronym relationship between "VM/SP" and "operating system" in the knowledge base, as shown in Figure 11. In this case, the co-occurrence consistency between "VM/SP" and "operating system" holds, since there is a relationship between the words and both cases are consistent (the grammatical case subject can have a semantic case agent), as shown in Figure 12. The value of co-occurrence consistency is the number of co-occurrence modifiers that are consistent between the path and the sentence. Here, the value is 1, since only one co-occurrence modifier "VM/SP" is consistent.

3. Context consistency

Context consistency holds if dependencies in a path already exist in previous sentences. For example, if the sentence "the data is stored in the storage device" comes before the above sentence, then the dependency structure shown in Figure 13 is in the context base where the dependency structures of previous sentences are stored. Then the other path (shown in Figure 14), which corresponds to the dependency between "store" and "disk" in the "path," is found by using the context base. Thus the dependency between "store" and "disk" is defined by the context. The value of context consistency is the number of dependencies in the path that are defined by the context. In this case, the value is 1, since there is one dependency in the path and it is defined in the context.

The dependency distance is computed from the following formula:

\[
\text{Distance} = \frac{|\text{Dep}| + \text{V}_{\text{Cont}} \times (n - 1)}{(\text{V}_{\text{Case}} + 1) \times (\text{V}_{\text{Cont}} + 1)},
\]

where \(|\text{Dep}|\) represents the number of dependencies included in the path, \(\text{V}_{\text{Case}}\) is the value of case consistency, \(\text{V}_{\text{Cont}}\) is that of co-occurrence consistency, and \(\text{V}_{\text{Cont}}\) is that of context consistency.

This formula assumes that case and co-occurrence consistency affect the distance of the whole path, but that context consistency affects the distance of each dependency in the path.

\(n\) is a real number in the range \(0 \leq n \leq 1\); it is a heuristic parameter that represents the degree of unimportance of context consistency.

The dependency distance between "keep" and "virtual disk" that is calculated by using the path in the example is 0.125, because the number of dependencies is 1, the value of case consistency is 1, that of co-occurrence consistency is 1, and the value of context consistency is 1 (\(n\) is defined as 0.5).

The ambiguity of an attachment is resolved by selecting the candidate dependency that is separated by the shortest distance.
Table 2: Constraint Tables

| Constraint Table T5.6 | Constraint Table T5.7 | Constraint Table T6.7 |
|----------------------|----------------------|----------------------|
| 1 1 1 2 0 1 0 1 1 1 1 0 1 1 1 | 5 1 3 0 5 0 3 0 5 0 3 0 5 0 3 0 | 3 1 1 2 0 3 1 1 2 0 5 1 1 2 0 5 |
| 2 2 C1 0 1 2 C1 1 2 C1 1 2 C1 1 2 C1 1 | 4 3 C1 0 4 3 C1 1 4 3 C1 1 4 3 C1 1 | 5 3 C7 1 5 3 C7 1 5 3 C7 1 |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

Figure 15: Ambiguous Dependency Structure

5 Planning, Constraint Propagation, and Process of Disambiguation

When there are several attachment ambiguities in one sentence, the relationships of each pair of ambiguities are represented by a constraint network [9]. The idea that ambiguous syntactic structures can be represented by a data structure of constraint network was originally developed by Hiroshi Maruyama [7]. A constraint network consists of constraint tables.

For example, the constraint tables shown in Table 2 are constructed from the ambiguous dependency structure shown in Figure 15. In this dependency structure, words 5, 6, and 7 have attachment ambiguities, so their possible modifiees are \{1,3\}, \{1,3\}, and \{3,6\} respectively. The constraint table is a two-dimensional matrix that represents the possibility of simultaneous modification of two ambiguous attachments. The rows and columns of the matrix show the candidate modifiees of each modifier, and an element in the matrix means the possibility (1 or 0) that both dependencies can exist simultaneously. For example, constraint table T5.7 indicates that if word 5 modifies word 1, then word 7 cannot modify word 3 because of the rule of no-crossing.

By using the constraint tables, the system decides which ambiguity should be resolved first. This process is called planning. In the above example, words 5, 6, and 7 have two candidate modifiees each. But from the constraint tables, we can see that if word 7 modifies word 3, then words 5 and 6 cannot modify word 1. Thus, in this case, the ambiguity concerning the modification of word 7 should be resolved first. The algorithm for planning consists of the following steps:

1. On each row of the constraint table T_{i,j}, sum up the element values \(A_i\) in Table 2, and subtract the sum from the size of the row \(B_i\). Then sum up the results on all rows \(C_i\). The result is the value of merit of the ambiguity of word \(i\).

2. Do the same in each column. The result is the value of merit of the ambiguity of word \(j\).

3. In all the constraint tables, sum up all the values of merit of each ambiguity, and divide each of these values by the number of their candidate modifiees.

4. The expected values of merit of all ambiguities are given by the above process. Select the ambiguity that has the highest expected value.

When an ambiguity is resolved, the system updates the constraint tables by the filtering algorithm called constraint propagation. We apply Mohr and Henderson's AC-4 algorithm [8] for constraint propagation. We reduce the computational cost of disambiguation by using planning and constraint propagation.

Structural disambiguation of a sentence is done as follows. The PEG parser parses a sentence and constructs its phrase structure. The Dependency Structure Builder translates the phrase structure into the dependency structure, and constructs the constraint tables when the phrase structure contains several structural ambiguities. The Planner, which is the component for planning, gives the Disambiguator the information on an ambiguous dependency and its candidate modifiees. The Disambiguator decides which modifiee is the most preferable by doing path search and distance calculation. After resolving one ambiguous attachment, it calls the constraint propagation routine to filter the other ambiguities' candidates. After filtering, the Transformer transforms the dependency structure into one that has correct dependencies for all resolved attachments. These processes are iterated until no ambiguity remains.

6 Related Work

There are several approaches to structural disambiguation, including resolution of prepositional phrase attachment. Wilks et al. [12] discussed some strategies for disambiguation based on preference semantics. Our framework is closely related to their ideas. While their strategies need hand-coded semantic formulas called preplates to decide preferences, our system can construct dependency knowledge semi-automatically. Dahlgren and McDowell [2] proposed another preference strategy for prepositional phrase disambiguation. It is based on ontological knowledge, which is manually constructed. Whereas this framework (and also that of Wilks et al.) was aimed at disambiguating single prepositional phrases in sentences, our approach can handle the attachments of multiple prepositional phrases in sentences. Hirst [3] developed a mechanism for structural disambiguation, called the Semantic Enquiry Desk, which is based on Chraniak's marker passing paradigm [1]. Our path search is partially equivalent to marker passing. While marker passing involves a high computational cost and finds many meaningless relations, our path search is restricted and finds only paths that include synonym/taxonym relationships and dependencies. Our system can reduce the computational cost by using a limited knowledge search. Jensen and Binot [6] developed a heuristic method of prepositional phrase disambiguation
using on-line dictionary definitions. Our approach is similar to theirs in the sense that both use dictionaries as knowledge sources. The differences are in the ways in which dictionary definitions are used. While their method searches for knowledge by phrasal pattern matching and calculates certainty factors by complex procedures, ours uses knowledge in a simple and efficient way, searching trees and traversing nodes, and calculates preferences by a few simplified processes. Wermter [11] proposed a connectionist approach to structural disambiguation of noun phrases. We integrated syntactic and semantic constraints on the relaxation network. Semantic constraints on prepositional relationships between words are learned by a back-propagation algorithm. Learned semantics is often very useful for natural language processing, when semantic relationships cannot be represented explicitly. We represent semantic relationships between words by explicit relationship chains, and therefore do not need learning by back-propagation. We integrate semantic preferences and syntactic constraints by using constraint propagation, but it is a sequential connection and does not allow their interaction. We are thinking of designing a framework that deals with both syntactic and semantic constraints simultaneously.

7 Concluding Remarks

We developed the Dependency Analyzer to resolve structural ambiguity by semantic processing. It aims to overcome two serious problems in resolving practical semantic processing: (semi-)automatic construction of knowledge and efficient use of that knowledge. The key ideas, path search and distance calculation, were shown to be feasible.

We now have a knowledge base constructed by using definitions given in the “IBM Dictionary of Computing,” which includes about 20,000 instances of dependency structures. In addition, we evaluated the system by disambiguating the propositional phrase attachment of about 2,000 sentences. The results were as follows: (1) the number of ambiguous propositional phrases was 4,290, (2) the number of correctly disambiguated attachments was 3,569, and (3) the success ratio of disambiguation was 83.2%.

Further enhancement plans are listed below:

- We are exploring the formalization of dependency distance with reference to graph theory. Dependency distance is assumed to be a score for the consistency of a dependency with the background knowledge and context. The background knowledge and context are represented as trees (special cases of graphs), and consistency might be defined by a degree of matching between trees.

- We are planning to enhance the system for other problems such as adverb attachment and scope of conjunctions. To resolve general structural ambiguity problems, we must design a general ambiguity-packed syntactic structure, since the system can deal with locally packed ambiguities.

Acknowledgements

I would like to thank members of the IBM Tokyo Research Laboratory, Karen Jensen of the IBM Thomas J. Watson Research Center, and the reviewers for their valuable comments on a draft of this paper. Hiroshi Nomiyama for his help in implementing the system, Mizuho Tanaka, Yoko Kobayashi, Mitsuyo Sadahara, and Tomoko Uchida for their kind support in constructing the knowledge base and evaluating the system, and Michael McDonald for his helpful advice on the wording of this paper.

References

[1] Charniak, E., “A Neat Theory of Marker Passing,” Proceedings of AAAI-86, 584–588, 1986.

[2] Dahlgren, K. and McDowell, J., “Using Commonsense Knowledge to Disambiguate Prepositional Phrase Modifiers,” Proceedings of AAAI-86, 590–593, 1986.

[3] Hirst, G., Semantic Interpretation and the Resolution of Ambiguity, Cambridge University Press, 1987.

[4] Jacobs, P. and Zernik, U., “Acquiring Lexical Knowledge from Text: A Case Study,” Proceedings of AAAI-88, 739–744, 1988.

[5] Jensen, K., Heidorn, G.E., Richardson, S.D., and Haas, N., “PLNLP, PEG, and CRITIQUE: Three Contributions to Computing in the Humanities,” IBM Research Report, RC 11841, 1986.

[6] Jensen, K. and Binot J-L., “Disambiguating Prepositional Phrase Attachments by Using On-Line Dictionary Definitions,” Computational Linguistics, 13:251–260, 1987.

[7] Maruyama, H., “Structural Disambiguation with Constraint Propagation,” Proceedings of the 28th Annual Meeting of the ACL, 1990.

[8] Mohri, R. and Henderson, T., “Arc and Path Consistency Revisited,” Artificial Intelligence, 28:225–233, 1986.

[9] Montanari, U., “Networks of Constraints: Fundamental Properties and Applications to Picture Processing,” Information Sciences, 7:95–132, 1974.

[10] Nakamura, J. and Nagao, M., “Extraction of Semantic Information from an Ordinary English Dictionary and its Evaluation,” Proceedings of COLING-88, 459–464, 1988.

[11] Wermter, S., “Integration of Semantic and Syntactic Constraints for Structural Noun Phrase Disambiguation,” Proceedings of IJCAI-89, 1486–1491, 1989.

[12] Wilks, Y., Huang, X., and Fase, D., “Syntax, Preferences and Right Attachment,” Proceedings of IJCAI-85, 779–781, 1985.