IEICE TRANS. INF. & SYST., VOL.E92–D, NO.10 OCTOBER 2009

PAPER

Dependency Parsing with Lattice Structures for Resource-Poor Languages

Sutee SUDPRASERT[†a)], **Asanee KAWTRAKUL**[†], ***Members***, **Christian BOITET**[††], ***and*** **Vincent BERMENT**[††], ***Nonmembers***

SUMMARY In this paper, we present a new dependency parsing method for languages which have very small annotated corpus and for which methods of segmentation and morphological analysis producing a unique (automatically disambiguated) result are very unreliable. Our method works on a morphosyntactic lattice factorizing all possible segmentation and part-of-speech tagging results. The quality of the input to syntactic analysis is hence much better than that of an unreliable unique sequence of lemmatized and tagged words. We propose an adaptation of Eisner's algorithm for finding the k-best dependency trees in a morphosyntactic lattice structure encoding multiple results of morphosyntactic analysis. Moreover, we present how to use Dependency Insertion Grammar in order to adjust the scores and filter out invalid trees, the use of language model to rescore the parse trees and the *k*-best extension of our parsing model. The highest parsing accuracy reported in this paper is 74.32% which represents a 6.31% improvement compared to the model taking the input from the unreliable morphosyntactic analysis tools.

key words: *dependency parser, under-resourced languages, morphosyntactic lattice structure, Dependency Insertion Grammar, k-best parsing*

1. Introduction

Dependency representations date back to Tesnière and have been used extensively in NLP by Western and Eastern European and Japanese research groups since the early 1960's, notably for Machine Translation (MT). Constituent or "phrase-based" representations have also been used, mainly because of their good formal characterization by context-free grammars (CFG), and the existence of polynomial all-path algorithms[*]. The constituent representations have been applied, primarily for other applications such as Natural Language (NL-)based information retrieval[**].

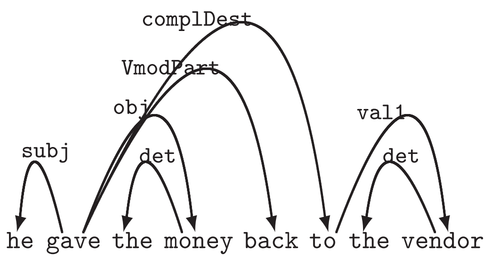
Enriched constituent trees were then produced by CFGs enriched by various means, such as attributes (used in MT at Grenoble since 1961), complex categories (GSPG [1]), feature structures (LFG [2], HPSG [3], and TAG [4]), and logical terms (metamorphosis grammars [5], DCG [6], and slot grammars for LMT [7]).

Mixed constituent and dependency tree representations have also been used and are used in NLP (Vauquois' multi-level structure, proposed in 1974 for MT, Starosta's lexicase trees, and HPSG).

Nowadays, projective and non-projective dependency structures have recently become quite fashionable in various NLP areas, such as MT [8], Information Extraction [9], Text Summarization [10], and Ontology [11].

There are several reasons for that:

- they are more economical (they have fewer nodes) and hence perspicuous than constituent structures;
- they can represent some discontinuous constituents in a projective way. For example, in "he gave the money back to the vendor", *gave... back* is a discontinuous constituent which cannot be represented by a constituent tree having a projective correspondence with the sentence. However, the following dependency tree is projective:



That is not always possible. For example, "Ces femmes, les hommes *ne* les ont *pas* encore tous comprises." ("These women, the men did not yet understand them$_{women}$ all$_{men}$.") has no reasonable projective dependency tree.
- they represent long-distance dependencies and predicate-argument relations (information needed in these applications) in a clearer way.

Figure 1 shows an example of discontinuous constituents represented by a dependency structure. In this case, there is no way to draw the correspondence lines between the nodes of the dependency tree and the words of the sentence (written as usual linearly) without any crossing pair of

Manuscript received October 27, 2008.
Manuscript revised June 13, 2009.

[†]The authors are with the NAiST (special research unit on NAtural language processing and Intelligent information System Technology), Department of Computer Engineering, Kasetsart University, Bangkok, Thailand.
[††]The authors are with the GETALP Group (Study Group on Machine Translation and Automated Processing of Languages and Speech), LIG (Grenoble Informatics Lab), Université Joseph Fourier (Grenoble 1), France.
a) E-mail: sutee.sudprasert@uknowcenter.org

DOI: 10.1587/transinf.E92.D.2122

[*]Notably one is CYK algorithm, for CFG in Chomsky normal form, and one is Earley, for any CFG.

[**]For example, in the LUNAR project around 1972-74, for which W. Woods and his team used ATNs.

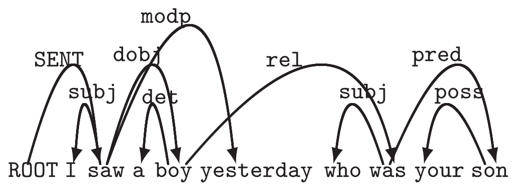


Fig. 1 Example of discontinuous constituents: the noun phrase "a boy who was your son" and the verb phrase "saw yesterday" are "shuffled".

lines[†]. We then say that the tree is "non-projective".

So far there have been two different approaches to analyze utterances into dependency structures: with or without an explicit Dependency Grammar (DG). One is rule-based [12]–[17], the rules being given manually, possibly augmented with probabilities. The problem with this approach is that it is difficult to write all rules manually. In addition, as the size grows, it becomes practically impossible to modify them to improve their quality in a systematic and monotonic fashion.

The other approach is corpus-based: knowledge is extracted automatically from annotated corpora and then transformed into parameters (i.e. probabilities or actions of shift-reduce parsers) for building the parser.

This approach has the potential to overcome many of the problems of the rule-based approach. Of course, a rich set of training data and accurate knowledge are crucial for this method. Various methods have been proposed for the learning part of this approach: learning actions of a deterministic parser [18], [19], learning similarity of tree structures [20], [21], and learning the scores of dependencies [22]–[24].

Among all these approaches there is one that tries to strike a balance between rule-based and corpus-based approaches. It infers parsing rules automatically on the basis of a tree-annotated corpus [25]. That approach has the advantage of both methods, allowing experts to refine and correct the learned rules in order to improve parsing accuracy. While our work falls into this category, we consider, in addition, scenarios without the use of high quality morphosyntactic analyzers or richly tree-annotated corpora. Obviously, in such a situation, one cannot assume the parser input to be a perfectly disambiguated string, neither can one expect high quality and very general rules.

In recent years, various researchers have started to build parsers with or without small annotated corpora. For example, Jinshan, et al. [26] proposed a method to build a dependency parser for Chinese. They used a purely statistical approach and relied only on part-of-speech information. Hwa, et al. [27] proposed a method for building a parser based on a parallel corpus, by inducing parse trees on the basis of parallel text. To do so, they need a parallel corpus and a reliable parser of the source language. However, all of them assume that the morphosyntactic analysis work properly.

Written Thai, the language we are working on, does not have word delimiters, and there are no reliable word segmenters and part-of-speech taggers for this language. Unlike many other languages, syntactic analysis of Thai should start from multiple results of a morphosyntactic lattice (see Fig. 2), rather than from a dubiously disambiguated string[††]. Previous work is generally based on the assumption that the input is a disambiguated morphosyntactic string. Unfortunately, this is not very realistic for a language like Thai with its potential ambiguities due to word segmentation and part of speech tagging. This is why we propose an extended parsing technique, able to handle an input as a morphosyntactic lattice, with multiple ways of segmentation and part-of-speech tagging.

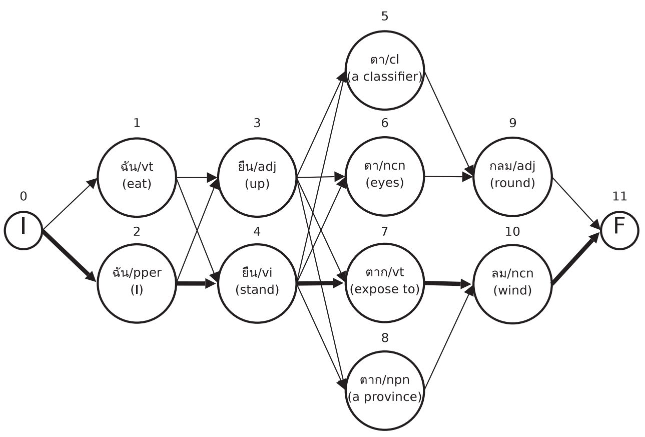


Fig. 2 The morphosyntactic lattice that encodes all possible word segmenting and part-of-speech tagging results of a Thai text that means "I stand (to) expose to (the) air" (the bold lines represent the correct result). Please see the abbreviation of part-of-speeches in Appendix A.1.

In fact, the idea of parsing multiple versions of a given sentence is not new. Since 1986, Tomita [28] proposed an efficient word lattice parsing algorithm that can be viewed as an extended LR parsing algorithm for context-free phrase structure grammars. Dependency parsing of word graphs has also already been proposed [29]. The parser was an extended version of the Constraint Dependency Grammar (CDG) parser developed by Maruyama [30]. It actually was used to eliminate multiple sentence hypotheses produced by the speech recognizer. The complexity of the parser is $O(n^4)$, where binary constraints were used (the complexity will increase with the number of constraints). In recent work, Collins, et al. [31] extended head-driven parsing models [32] to parse word lattices, in order to use as a simultaneous language model and a parser for large-vocabulary speech recognition. The system experimented on Wall Street Journal treebank shows better accuracy than the standard n-gram language model.

[†]The expression "tree without crossing lines" if often found in the literature but is faulty, as a tree can *always* be drawn on a plane without any crossing branches. The thing which is projective or not is the *correspondence* between the string and the tree, which should better be represented by "liaison elements", as in entity-relation diagrams.

[††]A dubiously disambiguated string means a string disambiguated by an unreliable morphosyntactic analyzer.

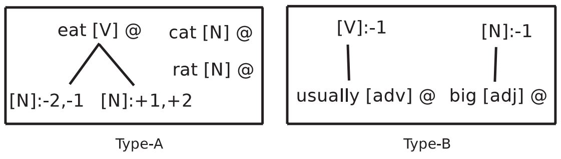


Fig. 3 Type-A and Type-B elementary trees. Note: the @ symbol represents the head node of the tree.

Those previous works intended to be used for improving the accuracy of continuous speech recognition. In this paper, we present a different method, a data-driven parser, based on Eisner's algorithm [33]. Using our algorithm, the time complexity of the parser is $O(n^3)$, the same as Eisner's. Moreover, our parser can be augmented with a Dependency Insertion Grammar (DIG) [34], increasing accuracy by eliminating illegal trees, which is useful when a training corpus is small.

The method we present here deals with the above-mentioned problems. First, we allow the input to be more than a single string. We propose a possible solution, parsing a morphosyntactic lattice. Second, we try to combine a "statistical" and an"expert" approach in order to optimize the parser performance. We also use the DIG which is automatically extracted from a tree-annotated corpus, and we propose methods used for rescoring sub-trees with a language model, which is necessary where the Eisner's algorithm is applied to parse a morphosyntactic lattice. Moreover, those extensions do not increase the time complexity, and the time complexity of k-best parsing is only increased by a multiplicative factor, $O(k \log k)$.

The rest of this paper is organized as follows. In the next section, we briefly describe the DIG and how to automatically acquire them from a tree-annotated corpus. Section 3 presents our dependency parsing algorithm which can handle a morphosyntactic lattice. In Sect. 4, we report on our experiments based on NAiST Dependency Treebank and analyze the results.

2. Dependency Insertion Grammar

2.1 Basis of DIG

The DIG formalism [34] consists of two important parts: elementary trees and insertion operation. Each node in an elementary tree consists of: a lexical item, a corresponding part-of-speech and a local word ordering. The elementary trees are of two types: Type-A and Type-B. The difference between them (see Fig. 3) is that the root node of Type-A is lexicalized and is the head of the tree, while the root of Type-B is not lexicalized but one of the lexicalized nodes is the head of the tree.

Insertion is the only operation used for DIG derivation: when an unlexicalized node of an elementary tree is of the same type, i.e. category, as the head node of another elementary tree, the two can be unified into a single node and a new elementary tree can be built (see Fig. 4).

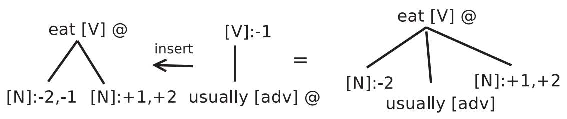


Fig. 4 The insertion operation in DIG where −2, −1, +1, and +2 are relative positions.

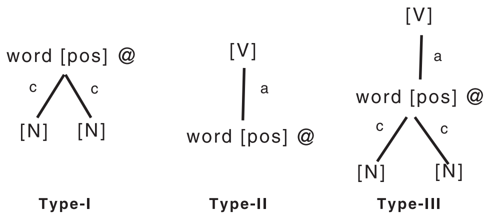


Fig. 5 Three forms of elementary trees. Note: the"c" and "a" functions stand respectively for "complement" and "adjunct".

2.2 The Extended Forms of Elementary Trees

The DIG formalism itself does not have any constraints concerning the shape of the elementary trees, as long as they satisfy the requirements of Type-A and Type-B elementary trees. Therefore, given a corpus, there can be a number of elementary trees, each of which covers the corpus. To say that some elementary trees cover the corpus implies that each dependency tree in the corpus can be generated by combining the elementary trees by the insertion operation.

Like Xia [35] who proposed a method for extracting LTAGs (Lexicalized Tree-Adjoining Grammars) from a bracketed corpus, we need to define extended forms of elementary trees in order to ensure that the extracted grammar is both compact and linguistically sound (the smallest set of grammars that can recognize all sentences in the training data.).

We consider extended elementary trees of the following forms (see Fig. 5):

- Type-I tree: a type A tree with only complement functions[†] (syntactic arguments, i.e. strongly bound complements).
- Type-II tree: a type B tree with only two nodes, the unique arc bearing an adjunct function (circumstantial complements).
- Type-III tree: a combination of Type-I and Type-II.

Given the fact that our tree-annotated corpus is very small, it is unavoidable to run into a data sparseness problem. In order to minimize this problem, we used relative direction instead of relative position i.e. position ≤ -1 be left(L) and position $\geq +1$ be right(R).

Figure 6 shows an elementary tree corresponding to "eat [VT]" (transitive verb). The left-hand side tree is a normal elementary tree of Type-I, waiting for "PPER" at posi-

[†]The grammatical functions used for annotating our treebank consist of 30 types that can be divided into 2 main types, i.e. complement and adjunct.

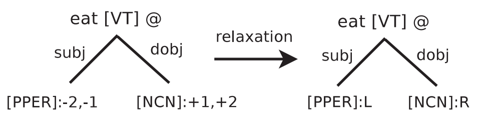


Fig. 6 An example of the relaxation of DIG.

Algorithm 1 *ExtractDIG(T)*

```
Input: $T$ a parse tree object
Output: a set of elementary trees
 1: $ElementaryTrees \leftarrow [\,]$
 2: for all $x$ in $T.nodes()$ do
 3:     $T_1 \leftarrow new\_tree(x.word, x.pos)$
 4:     for all $y$ in $x.children()$ do
 5:         if $y.function$ = 'c' and $x.function$ = 'c' then
 6:             $T_1.connect\_at\_root(null, y.pos)$
 7:         else if $y.function$ = 'a' then
 8:             if $y.children() = [\,]$ then
 9:                 $T_2 \leftarrow new\_tree(null, x.pos)$
10:                 $T_2.connect\_at\_root(y.word, y.pos)$
11:                 $ElementaryTrees.append(T_2)$
12:             else
13:                 $T_3 \leftarrow new\_tree(null, x.pos)$
14:                 $tmp \leftarrow new\_tree(y.word, y.pos)$
15:                 for all $z$ in $y.children()$ do
16:                     if $z.label$ = 'c' then
17:                         $tmp.connect\_at\_root(null, z.pos)$
18:                     end if
19:                 end for
20:                 $T_3.connect\_at\_root(tmp)$
21:                 $ElementaryTrees.append(T_3)$
22:             end if
23:         end if
24:     end for
25:     $ElementaryTrees.append(T_1)$
26: end for
27: return $ElementaryTrees$
```

tion -1 and "NCN" at position $+1$. When the constraints are relaxed, the elementary tree is changed to the right-hand side tree.

2.3 Extracting Elementary Trees from the Treebank

Psuedo-code of the algorithm shown in Algorithm 1 is used for extracting elementary trees of the above three mentioned forms from the treebank. The algorithm starts from the root node and traverses each node in order to recognize the pattern of the extended form.

Given a training dependency tree, the algorithm traverses each node of the tree (line 2). At any visited node x, if inward edge is complement function (c) then a Type-I tree will be built where the visited node is the root and the part-of-speech of its children which have complement function are Type-I tree's children (line 3-6, 25). If an adjunct relation (a) with an outward edge is found, there are two possible cases: if a child of a visited node y doesn't have any children, a Type-II tree will be built immediately by using the part-of-speech of the visited node x as the root and its child node y as the child (line 8-11). Otherwise, if a child of the visited node y has children, the algorithm will do like constructing a Type-I tree in order to produce a subtree (line 14-19). A Type-III tree will be built by using the part-of-speech of the visited node x as the root, connecting the produced sub-tree *tmp* to the root. If the child of the visited node y does not have any children, a Type-II tree will be built instead (line 20-21).

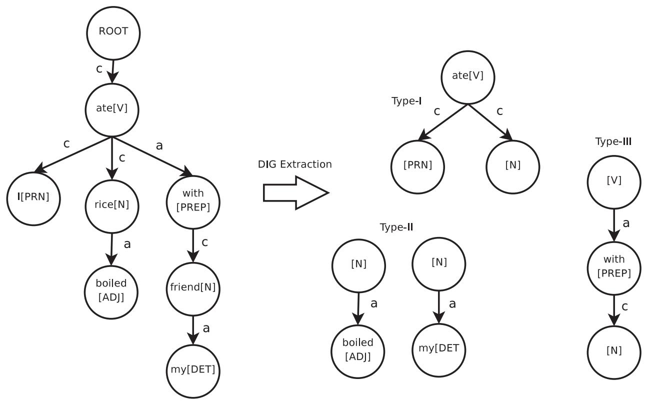


Fig. 7 An example of extracting the extended forms of elementary trees from a parse tree.

Figure 7 shows an example of DIG elementary trees extracted from the annotated-tree text "I ate boiled rice with my friend".

3. Dependency Parsing Technique

The dependency parsing technique we use is based on the assumption that the score (probability) of a dependency tree is the sum of the scores of all edges in the tree, and that the best parse tree is the one with the highest score. This approach consists of two processes: searching the best tree among all possible trees and computing the scores of edges.

3.1 Searching the Best Tree

3.1.1 Eisner's Algorithm

In the traditional approach, the input is a disambiguated morphosyntactic string $w_1 \ldots w_N$ where N is the number of words in the string. A $N \times N$ Dependency Matrix, DM, is built, where $DM[i, j]$ contains the best relation rel between w_i and w_j, with a score s.

$$DM[i, j] = \begin{cases} (rel, s) & \text{if } i \neq j \\ \text{empty} & \text{otherwise} \end{cases} \tag{1}$$

We can apply Eisner's algorithm [33] [†] to find the best tree from all possible results in the dependency matrix within $O(n^3)$. The Eisner's algorithm is a bottom-up parsing algorithm just like the CKY parsing algorithm: it finds optimal subtrees for substrings of increasing length. The idea is to parse the left and right dependents of a word independently, and combine them later. That requires only two additional binary variables to specify the direction of the item

[†]See [24], [33] for more details.

Algorithm 2 Eisner's algorithm

```
Input: $DM$ a dependency matrix size $n \times n$; $n > 1$
Output: a score of the best parse tree
 1: for all $p, d, c$ in $\{1..n\} \times \{\Leftarrow, \Rightarrow\} \times \{0, 1\}$ do
 2:     $C[(p, p, d, c)] \leftarrow 0.0$
 3: end for
 4: for $m = 1$ to $n$ do
 5:     for $p = 1$ to $n - m$ do
 6:         $t \leftarrow p + m$
 7:         for $r = p$ to $t$ do
 8:             if $r < t$ then
 9:                 $C[(p, t, \Leftarrow, 0)] \leftarrow max(C[(p, t, \Leftarrow, 0)], C[(p, r, \Rightarrow, 1)] + C[(r + 1, t, \Leftarrow, 1)] + DM[t, p])$ {create left incomplete items}
10:                 $C[(p, t, \Rightarrow, 0)] \leftarrow max(C[(p, t, \Rightarrow, 0)], C[(p, r, \Rightarrow, 1)] + C[(r + 1, t, \Leftarrow, 1)] + DM[p, t])$ {create right incomplete items}
11:                 $C[(p, t, \Leftarrow, 1)] \leftarrow max(C[(p, t, \Leftarrow, 1)], C[(p, r, \Leftarrow, 1)] + C[(r, t, \Leftarrow, 0)])$ {create left complete items}
12:             else if $r > p$ then
13:                 $C[(p, t, \Rightarrow, 1)] \leftarrow max(C[(p, t, \Rightarrow, 1)], C[(p, r, \Rightarrow, 0)] + C[(r, t, \Rightarrow, 1)])$ {create right complete items}
14:             end if
15:         end for
16:     end for
17: end for
18: return $C[(1, n, \Rightarrow, 1)]$
```

and whether the item is complete, i.e., d and c which will be explained more in the following.

In the dynamic programming table C of Eisner's algorithm, $C[(p, t, d, c)]$ stores the score of the best subtree from position p to position t, with direction d and complete value c. The variable d indicates the direction of the subtree (whether it gathers left ($\Leftarrow$) or right ($\Rightarrow$) dependents). The variable c indicates whether a subtree is complete ($c = 1$, no more dependents) or not ($c = 0$, needs to be completed).

The pseudo-code of Eisner's algorithm[†] is shown in Algorithm 2. Consider the line 9 in Algorithm 2. It says to find the best score for an incomplete left subtree from position p to t, $C[(p, t, \Leftarrow, 0)]$. We need to find the index $p \leq r < t$ that gives the best (maximum) possible score for combining two complete subtrees, $C[(p, r, \Rightarrow, 1)]$ and $C[(r+1, t, \Leftarrow, 1)]$. The score of combining these two complete trees is the score of these subtrees plus the score of a dependency relation from w_i to w_j. This is guaranteed to be the score of the best subtree because we are considering all possible combinations by enumerating over all values of r. By forcing a root node at the left-hand side of the sentence, the score of the best tree for the sentence is $C[(1, n, \Rightarrow, 1)]$.

3.1.2 Extended Parsing Technique for Handling Lattice Structure

In order to extend parsing technique for handling lattice structure, we still use exactly the same Dependency Matrix as in (1), but adding one more condition to check whether there is a non-empty path from w_i to w_j in the lattice, which is acc(i, j) in (2). Algorithm 3 shows the adaptation of Eisner's algorithm to find the projective dependency trees.

$$DM\,[i, j] = \begin{cases} (rel, s) & \text{if } i \neq j \text{ and acc}(i, j) \\ \text{empty} & \text{otherwise} \end{cases} \qquad (2)$$

The modification to Eisner's original algorithm consists just in adding a condition *IsLegal* (line 9, 14 and 17 in Algorithm 3) and a function *get_next* (in line 8) to validate the built subtrees along the lattice structure. The call *get_next*(n) returns all next adjacent nodes of n, and

$$IsLegal(p, r, q, t) = acc(p, r) \wedge acc(q, t) \qquad (3)$$

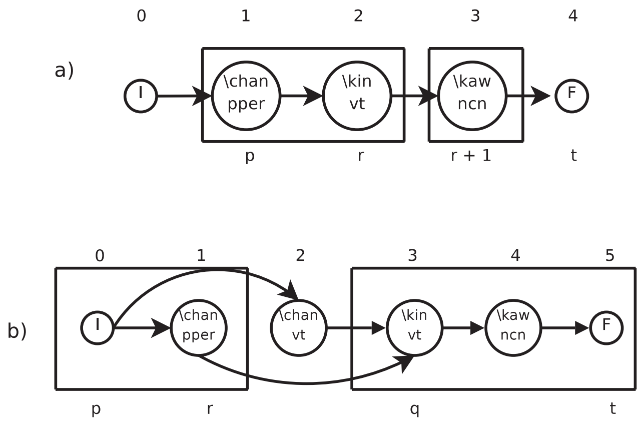


Fig. 8 An example of parsing with a) a correct morphosyntactic analyzed text and b) a morphosyntactic lattice.

In order to illustrate this, let's take a look at Fig. 8, showing the parsing processes respectively of Eisner's algorithm (Algorithm 2) and ours (Algorithm 3).

Figure 8 a) shows how to find the best incomplete subtree corresponding to the sub-string $p \ldots t$ of Eisner's algorithm. The best incomplete sub-tree for sub-string $p \ldots t$ is the combination of sub-trees $p \ldots r$ and $r + 1 \ldots t$ plus a dependency between w_p and w_t, which have received the highest score when $p \leq r < t$.

If the input is a morphosyntactic lattice, not every sub-

[†]The pseudo-code shows only computing the score of the best parse tree. We must also store back pointers so that it is possible to reconstruct the best tree from the chart item that spans the entire sentence. For simplicity, we assume that the output is an unlabeled dependency tree, therefore DM contains only scores.

Algorithm 3 Modified Eisner's algorithm for handling a lattice input

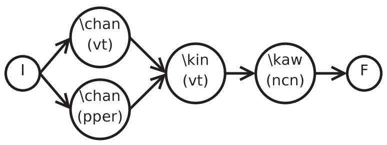
```
Input: $DM$ a dependency matrix size $n \times n$; $n > 1$
Output: a score of the best parse tree
1: for all $p, d, c$ in $\{1..n\} \times \{\Leftarrow, \Rightarrow\} \times \{0, 1\}$ do
2:     $C[(p, p, d, c)] \leftarrow 0.0$
3: end for
4: for $m = 1$ to $n$ do
5:     for $p = 1$ to $n - m$ do
6:         $t \leftarrow p + m$
7:         for $r = p$ to $t$ do
8:             for all $q$ in $get\_next(r)$ do
9:                 if $r < t$ and $IsLegal(p, r, q, t)$ then
10:                    $C[(p, t, \Leftarrow, 0)] \leftarrow max(C[(p, t, \Leftarrow, 0)], C[(p, r, \Rightarrow, 1)] + C[(q, t, \Leftarrow, 1)] + DM[t, p])$
11:                    $C[(p, t, \Rightarrow, 0)] \leftarrow max(C[(p, t, \Rightarrow, 0)], C[(p, r, \Rightarrow, 1)] + C[(q, t, \Leftarrow, 1)] + DM[p, t])$
12:                end if
13:            end for
14:            if $r < t$ and $IsLegal(p, r, r, t)$ then
15:                $C[(p, t, \Leftarrow, 1)] \leftarrow max(C[(p, t, \Leftarrow, 1)], C[(p, r, \Leftarrow, 1)] + C[(r, t, \Leftarrow, 0)])$
16:            end if
17:            if $r > p$ and $IsLegal(p, r, r, t)$ then
18:                $C[(p, t, \Rightarrow, 1)] \leftarrow max(C[(p, t, \Rightarrow, 1)], C[(p, r, \Rightarrow, 0)] + C[(r, t, \Rightarrow, 1)])$
19:            end if
20:        end for
21:    end for
22: end for
23: return $C[(1, n, \Rightarrow, 1)]$
```

tree can be combined to generate a new sub-tree. Only a combination which is in a *trajectory*[†] can be generated. By looking at Fig. 8 b) we can see that the selectable sub-strings need to contain a link between a starting node and an ending node. For example, sub-string $1 \ldots 2$ cannot be selected, because there is no path from 1 to 2 (validated by $IsLegal$ condition). Moreover, when we try to combine two sub-trees, we also have to check that there is an arc connecting the last node of the first sub-tree, r, to the first node of the second sub-tree, q (limited by *get_next* function).

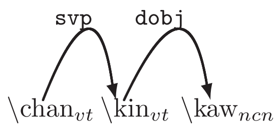
The computing time is increased by testing the condition and looking for all nodes q directly connected from r. Both can be done in constant time equaling the branching factor[††]. Hence we are still in $O(n^3)$ where n is the number of nodes in the lattice. Hence, there is no increase in time complexity.

3.2 Computing the Scores of Edges

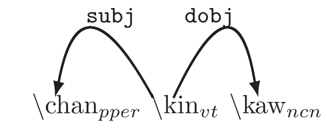
In fact, a function for computing the scores of edges can easily be estimated by using machine learning models such as Maximum Entropy Models or SVM. But in our case that a training corpus is very small, the use of machine learning model alone may leads the parser produces invalid parse trees. In addition, we assumed that dependencies in a sentence are independent of each other, hence the score of a dependency tree is the sum of the scores of all its edges. However, relying solely on the score of edges is not appropriate to compute the score of dependency trees for each possible output from a morphosyntactic analyzer (which are encoded in a lattice structure). Because it is possible that the parser will select a parse tree that has the highest score but the parse tree is not necessary to be a correct morphosyntactic analysis. See an example of parsing a simple sentence "\chan(I) \kin(eat) \kaw(rice)". It can be encoded into a morphosyntactic lattice



that \chan has two possible part-of-speeches i.e. personal pronoun (pper) and transitive verb (vt) while \kin and \kaw have one possibility, transitive verb and common noun (ncn) respectively. In this context, \chan should be personal pronoun. But the parser will produce



instead of



Since, in Thai, subjects are often omitted that makes the score of the first word being transitive verb and root of a

[†]We call *trajectory* of such any sequence of directly linked vertices in a lattice beginning with I and ending with F. The p_{th} *trajectory* has the form $T_p = I \rightarrow w_{p1} \rightarrow \ldots \rightarrow w_{pl_p} \rightarrow F$.

[††]Maximum number of outward edges of each node in the lattice.

Algorithm 4 *K*-best version of the modified Eisner's algorithm for lattice structure combining with DIG and language model

```
Input: $DM$ a dependency matrix size $n \times n$; $n > 1$, $k \geq 1$, $b \geq 0$, and $e > 0$
Output: a list of $k$-best parse tree
 1: for all $p, d, c$ in $\{1..n\} \times \{\Leftarrow, \Rightarrow\} \times \{0, 1\}$ do
 2:   $C[(p, p, d, c)] \leftarrow [\ ]$
 3:   for $i = 1$ to $k$ do
 4:     $C[(p, p, d, c)].append(\text{new } TreeObject())$
 5:   end for
 6: end for
 7: for $m = 1$ to $n$ do
 8:   for $p = 1$ to $n - m$ do
 9:     $t \leftarrow p + m$
10:     for $r = p$ to $t$ do
11:       for all $q$ in $get\_next(r)$ do
12:         if $r < t$ and $IsLegal(p, r, q, t)$ then
13:           $C[(p, t, \Leftarrow, 0)] \leftarrow merge_{\leq k}(C[(p, t, \Leftarrow, 0)], dig\_mult_{\leq k}(C[(p, r, \Rightarrow, 1)], C[(q, t, \Leftarrow, 1)], DM[t, p], b, e)$
14:           $C[(p, t, \Rightarrow, 0)] \leftarrow merge_{\leq k}(C[(p, t, \Rightarrow, 0)], dig\_mult_{\leq k}(C[(p, r, \Rightarrow, 1)], C[(q, t, \Leftarrow, 1)], DM[p, t], b, e)$
15:         end if
16:       end for
17:       if $r < t$ and $IsLegal(p, r, r, t)$ then
18:         $C[(p, t, \Leftarrow, 1)] \leftarrow merge_{\leq k}(C[(p, t, \Leftarrow, 1)], dig\_mult_{\leq k}(C[(p, r, \Leftarrow, 1)], C[(r, t, \Leftarrow, 0)], 0, b, e)$
19:       end if
20:       if $r > p$ and $IsLegal(p, r, r, t)$ then
21:         $C[(p, t, \Rightarrow, 1)] \leftarrow merge_{\leq k}(C[(p, t, \Rightarrow, 1)], dig\_mult_{\leq k}(C[(p, r, \Rightarrow, 0)], C[(r, t, \Rightarrow, 1)], 0, b, e)$
22:       end if
23:     end for
24:   end for
25: end for
26: return $C[(1, n, \Rightarrow, 1)]$
```

sentence very high . But if we look in morphosyntactical context, "\chan$_{vt}$ \kin$_{vt}$ \kaw$_{ncn}$" is invalid (The abbreviation of grammatical functions is given in Appendix A.2).

In order to surpass these problems, we will introduce two more methods for computing a score of a parse tree: adjusting the score of edges computed and filtering out invalid trees by applying the DIG and the rescoring sub-trees by using a Language Model (LM).

In addition, as our parsing algorithm is inspired by Eisner's algorithm allows for k-best extensions, we can also extend our adapted algorithm to compute the k-best trees. With the k-best extension, if the function f that computes a new score by merging two sub-trees is monotonic, the complexity of the parsing algorithm will be increased by a multiplicative factor, $O(k\ log\ k)$ [36].

We will present a method that efficiently computes the score of the best tree for the morphosyntactic lattice, and is a monotonic function. That will increase the time complexity of k-best parsing only by a multiplicative factor $O(k \log k)$.

The pseudo-code of the complete parsing algorithm is shown in Algorithm 4. In the algorithm, there are three parts that is different to Algorithm 3. First, items of table C is a list of $TreeObject$ containing a tree structure with its score (line 2-5). Second, $dig\_mult_{\leq k}$ is used to find k-best trees of all multiplication between two lists of trees (line 13,14,18,21). Third, we replace max with $merge_{\leq k}$ (line 13,14,18,21). The detail of $dig\_mult_{\leq k}$ and $merge_{\leq k}$ will be described in the Sects. 3.2.2 and 3.2.3 respectively.

3.2.1 The Scores of the Edges

The scores of edges are a score measuring the probability of the dependency relations established between two words. This score has been used in many researches and can be computed in various ways, for example, by using machine learning methods such as Maximum Entropy Models [22], Support Vector Machines [23], and MIRA [24] or conditional probabilistic models [26], [33], to estimate the score from kinds of linguistic features related with the two words.

In our work, we used Maximum Entropy Models for learning the score s. The features for training the model used here are similar to the first-order features used in [24]. But, we added more back-off features by adding a new tag set which is a POS (part-of-speech) generalization. For example, the seven tags of noun i.e., 'NCN', 'NCT','NNUM', 'NORM', 'NPN', 'NTIT', and 'NLAB', and a personal pronoun, 'PPER' will be reassigned to 'N'. The generalized part-of-speech has 18 different tags. Moreover, we discard 5-gram prefix feature[†] because it does not appropriate for Thai[††]. The model was used to estimate dependency probabilities. These probabilities will then be used as scores. In the implementation, we take the logarithm of the probabilities to avoid floating overflow.

[†]The 5-gram of a surface word will used as a feature if the word is longer than 5 characters, for instance the 5-gram feature of "general" is "gener".

[††]Thai is isolating languages and consists of more than 60 characters (excluding symbolic characters). Therefore it is highly possible that words having the same 5-gram prefix are not related.

We use Maximum Entropy Modeling Toolkit for Python[†] for implementing the computing score model. The model is a two-class classifier, deciding whether a pair of words should have a dependency relation or not. In this work, we focus primarily on unlabeled dependencies, but the model can be extended to assign grammatical functions to the dependency structure by using a single-stage (joint labeling) or two-stage method (see [24] for more detail).

3.2.2 Adjusting the Score of Edges and Filtering Out Invalid Trees

Since, our corpus is small (resource-poor languages), the use of the scores of edges mentioned above (in Sect. 3.2.1) is not enough. This work, then, propose the use of DIG for adjusting the scores. We adjust the scores of edges by checking whether the tree satisfies a given DIG. If it does not, the score of the edge used to build the tree is decreased and if the number of unsatisfactoriness is greater than a constant b, the tree will be filtered out. In order to filter out the invalid tress, we build larger subtrees and derive their corresponding elementary trees as the same time. If the elementary trees cannot be derived, we will adjust the score of the built dependency trees. The score will be decreased by a positive constant e if the insertion of two smaller elementary trees fails. An example of adjusting the score of edges is shown in Fig. 9.

Figure 9 gives an example of computing the score of a new subtree where there are two possibilities to build a new subtree from $T[1-2]$ and $T[3-3]$: connecting "I" to "fast" and "run" to "fast" (see 9a). But, the insertion between their elementary trees satisfies only the connection from "run" to "fast" (as 9b), hence the score of the tree constructed from connecting "fast" to "I" is decreased by the constant e (as 9c), while the score of connecting "fast" to "run" is computed normally (as 9d).

If all possible elementary trees corresponding to each substring are kept, the time complexity of the parsing algorithm will be $O(g^n n^3)$, where g is the maximal number of corresponding elementary trees per a smallest substring (a lexicon). In fact, we do not need to keep all possible complete elementary trees. We can keep only the status of the elementary trees corresponding to each word, since the parsing algorithm considers only building dependency relation between two words.

In fact, when the insertion is performed, only the category and relative position of the elementary tree of the inserted tree are considered. Let us consider again at Fig. 9 b. It shows that the insertion of "I" and "run" into elementary trees correspond to "fast". We do not need to know the elementary trees of "I run". We consider only what the elementary trees of "fast" are waiting for (in this case "fast" is waiting for "V"). Therefore, the time complexity becomes $O(gn^3)$.

The operation of adjusting the scores of edges and the filtering out the invalid trees are embedded into the operation $dig\_mult_{\leq k}$ (see Algorithm 5) i.e. the modified version

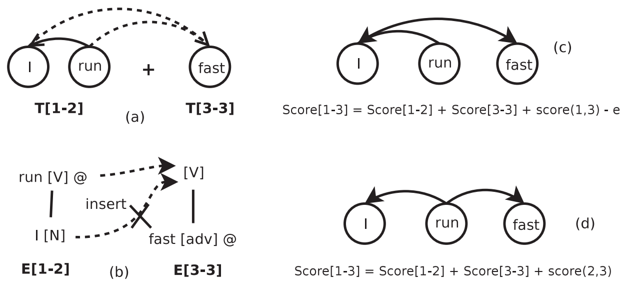

I
run
+
fast
T[1-2]
(a)
T[3-3]
I
run
fast
(c)
Score[1-3] = Score[1-2] + Score[3-3] + score(1,3) - e
run [V] @
[V]
insert
I [N]
fast [adv] @
E[1-2]
(b)
E[3-3]
I
run
fast
(d)
Score[1-3] = Score[1-2] + Score[3-3] + score(2,3)

Fig. 9 An example of adjusting the scores of edges by using DIG for "I run fast".

Algorithm 5 $dig\_mult_{\leq k}(C_1, C_2, score, b, e)$

```
Input: $C_1$ and $C_2$ are two lists of $TreeObjects$, $k > 0$,
    $score \geq 0$, $b \geq 0$, and $e > 0$
Output: a list of $TreeObjects$
 1: $results \leftarrow [\,]$
 2: for all $t_1,t_2$ in $mult_{\leq k}(C_1, C_2)$ do
 3:     if $badness(t_1, t_2) \leq b$ then
 4:         $tree \leftarrow combine(t_1, t_2)$
 5:         if $check\_dig(t_1, t_2)$ then
 6:             $tree.score \leftarrow t_1.score + t_2.score + score - e$
 7:         else
 8:             $tree.score \leftarrow t_1.score + t_2.score + score$
 9:         end if
10:         $results.append(tree)$
11:     end if
12: end for
13: return $results$
```

of $mult_{\leq k}$ [36], the multiplication operation that produces k-best trees of all multiplication between two lists of trees. The time complexity of the operation $mult_{\leq k}$ is $O(k \log k)$.

In Algorithm 5, the $badness(t_1, t_2)$ function returns the number of dependencies that do not satisfy the DIG if t_1 and t_2 are combined, and $check\_dig(t_1, t_2)$ returns $true$ if combination of t_1 and t_2 is satisfied by DIG, otherwise $false$. Finally, $combine(t_1, t_2)$ returns a new sub-tree which is a combination of t_1 and t_2. The b[††] constant is used for limiting the value "*badness*" (the number of dependencies not satisfying DIG) that is allowed to occur in the parse trees.

By adding the operation of adjusting the score of edges and filtering out invalid trees processes into the k-best parsing, the multiplicative factor is still $O(k \log k)$. However, the best sub-tree is no longer optimal because the penalization will be only applied to the k-best sub-trees in each step, so that it is possible that an other sub-tree that is not penalized will have higher scores than a sub-tree in the k-best list which is penalized. Therefore, the value of k has an effect on the parsing accuracy: if we increase k, the search space of finding the best sub-trees is also increased. Even though this method does not guarantee the optimum solution, the parsing accuracy can be improved with a small k (see Sect. 4.2 for the detail of the experiments).

[†]http://www.homepages.inf.ed.ac.uk/s0450736/maxent_toolkit.html

[††]The b will be increased dynamically by one if any parse tree cannot be generated.

Algorithm 6 $merge_{\leq k}(C_1, C_2)$

```
Input: $C_1$ and $C_2$ are two lists of $TreeObject$s and $k > 0$.
Output: a list of $TreeObject$s
 1: $R \leftarrow [\,]$
 2: $L_1 \leftarrow rescore\_by\_lang\_model(C_1)$
 3: $L_2 \leftarrow rescore\_by\_lang\_model(C_2)$
 4: while $len(L_1) > 0$ and $len(L_2) > 0$ and $len(R) < k$ do
 5:     if $L_1.first \geq L_2.first$ then
 6:         $R.append(L_1.first)$
 7:         $L_1.remove(L_1.first)$
 8:     else
 9:         $R.append(L_2.first)$
10:         $L_2.remove(L_2.first)$
11:     end if
12: end while
13: while $len(L_1) > 0$ and $len(R) < k$ do
14:     $R.append(L_1.first)$
15:     $L_1.remove(L_1.first)$
16: end while
17: while $len(L_2) > 0$ and $len(R) < k$ do
18:     $R.append(L_2.first)$
19:     $L_2.remove(L_2.first)$
20: end while
21: return $rescore\_by\_edges\_score(R)$
```

3.2.3 Rescoring Sub-Trees by Using Language Model

Rescoring sub-trees by using a LM is very important for finding the best parse tree from all possible segmented words and part-of-speech tagged sentences, since the use of the score of dependencies does not guarantee that the tree with the highest score will correspond to the best word segmented and best part-of-speech tagged sentence (as mentioned in Sect. 3.2). We use a LM to help the parser select a good parse tree which is also a good morphosyntactically analyzed text. Here, we use trigram model on part-of-speech to represent the LM.

$$P(t_{1,n}) = \prod_{i=1}^{n} P(t_i|w_i)P(t_{i-2}, t_{t-1}|t_i) \tag{4}$$

where t_i and w_i are part-of-speech and word at position i in a given sentence respectively. n is a length of the sentence.

In this work, we use the trigram model because it is fast, simple and easy to implement. Other methods could also give us a reasonable score of morphosyntactic analyzed input, such as Conditional Random Fields, that is theoretically better than the trigram model, but more complex to implement.

The trigram model was trained on 40,494 part-of-speech tagged sentences containing 49 tags, 18,396 words and 548,431 tokens. Table 1 shows the detail of the training corpus related to the part-of-speech ambiguity that we found.

The accuracy of the word segmenter and of the part-of-speech tagger by using the trigram model is about 95% and 90% respectively.

We added a rescoring process into the function $merge_{\leq k}$, which takes two sorted lists of length k (or fewer) as input, and outputs the top k in sorted order of the $2k$ elements. For parsing a single input, the elements (sub-trees) are sorted by the score of edges, but here, for parsing a morphosyntactic lattice, the elements are sorted by the score of LM instead. This can be done in $O(k \log k)$ then the overall multiplicative factor ($dig\_mult_{\leq k}$ and $merge_{\leq k}$ operations) is still $O(k \log k)$. The pseudo-code is shown in Algorithm 6.

In fact, the rescoring process can be added into the final parse trees, but we may have to set k extremely high in order to find the true best parsing (taking the LM into account).

Table 1 The ambiguity of the training corpus for LM.

ambiguity	words	tokens
1	16,262 (88.40%)	239,810 (43.73%)
2	1,789 (9.73%)	144,429 (26.34%)
3	270 (1.47%)	94,036 (17.15%)
4	56 (0.30%)	28,241 (5.15%)
5	14 (0.08%)	19,919 (3.63%)
6	5 (0.03%)	21,996 (4.01%)

4. Experiments

4.1 NAiST Dependency Treebank

The NAiST Dependency Treebank is the first Thai dependency treebank. It contains 816 sentences collected from various domains such as agricultural news, encyclopedia, and healthcare. Out of them, about 500 are aligned with English sentences, and they are useful resources for machine translation.

All of the parse trees in the treebank have projective dependency structures, because Thai is a strongly projective language. Non-projective structures rarely appear in writing text, although, they appear quite often in spoken dialogues.

For preparing the treebank, sentences were first segmented into words and part-of-speech tagged. This tagging comprised 49 categories and was done automatically by t3 tagger[†] trained on our part-of-speech tagged corpus. Then, the sentences were uploaded to Tred[††] [37], a web-based tree editing component, and then experts will correct and annotate them. We used 30 grammatical functions (12 complements and 18 adjuncts) for labelling the dependency relations.

4.2 Results and Analysis

To evaluate our methods, we set up three experiments. In the first, we assume that inputs are correctly word segmented and part-of-speech tagged. In this experiment, we can straight compare our method to the others. In the second experiment, we will not assume that the inputs are perfect, but we will convert the inputs into morphosyntactic lattices to test our method. For the others, the inputs will be analyzed by the existing morphosyntactic tools. Finally, we

[†]http://acopost.sourceforge.net

[††]http://naist.cpe.ku.ac.th/tred

study the oracle parse [38], or the best parse among the top k parses in order to measure the performance of the k-best parsing.

For the experiments, we used 716 sentences of the NAiST treebank for training and the other 100 sentences for testing. We measure Dependency Precision (DP), Complete Rate (CR) and Root Accuracy (RA) to evaluate the parsing results. We measured only the correctness of the dependency structures without considering the grammatical functions.

$$\mathrm{DP} = \frac{\text{number of correct dependencies}}{\text{total number of reference dependencies}}$$
$$\mathrm{CR} = \frac{\text{number of complete parse trees}}{\text{total number of sentences}}$$
$$\mathrm{RA} = \frac{\text{number of correct root nodes}}{\text{total number of sentences}}$$

In the second and third experiments, we also measured the correctness of the morphosyntactic analysis. We used Token Precision (TP), Token Recall (TR), and Token F-measure (TF).

$$\mathrm{TP} = \frac{\text{number of correct tokens}}{\text{total number of reference tokens}}$$
$$\mathrm{TR} = \frac{\text{number of correct tokens}}{\text{total number of hypothesis tokens}}$$
$$\mathrm{TF} = \frac{2 \times \mathrm{TP} \times \mathrm{TR}}{\mathrm{TP} + \mathrm{TR}}$$

Here, the number of correct tokens means the tokens that are correctly word-segmented and part-of-speech tagged.

4.2.1 Extracted Elementary Trees

The extended forms of elementary trees were extracted from the training corpus, 716 sentences, there are 8,172 tokens, of which 1,996 are type-I, 4,342 are type-II and 1,834 are type-III.

In the elementary trees extracted using the algorithm described in Sect. 2.3, there are 981 different types of elementary trees (175, 400 and 406 for Type-I, Type-II and Type-III, respectively), and 497 of them appear only once. Some of these elementary trees are abnormal structures especially those of low number of occurrence in the corpus.

Obviously, the extracted elementary trees do not cover all the words. Therefore, if the lexicalized elementary trees (the head is a lexicon with its part-of-speech) can not be found, the unlexicalized elementary trees (the head is a part-of-speech) will be matched instead. We allow specifically the unlexicalized elementary trees which have a high number of occurrence in the corpus (> 3) will be used in order to avoid using noisy elementary trees.

4.2.2 Parsing with Perfect Inputs

Although parsing with the perfect input is not the main focus of this work, observing the accuracy of parsing with a perfect input can help investigate the performance of combining DIG with a data-driven parsing method, and also with other parsers. Here, we use the MSTparser[†], a statistical dependency parser freely available on the web. The MSTparser was trained with the parameters, $k = 5$ and $N = 10$ that was reported [24] yielding a good accuracy and training the model in reasonable time.

Table 2 Results comparing our systems with MSTparser where the input is perfect (for MAX_{DIG}, we set $k = 1$, $b = \infty$ and $e = 2$).

	DP	**CR**	**RA**
MAX (baseline)	86.03	21.00	90.00
MST_{proj}	83.40	15.00	**94.00**
MAX_{DIG}	**88.66**	**27.00**	92.00

For our parsing algorithm, there are three parameters which can affect the performance i.e. k, e, and b. Therefore, we set up another experiment for observing the effect of these parameters by letting them vary.

Having experimented, we found that increasing k does not improve the accuracy. Moreover, the accuracy dropped at some higher k. The idea of using b does not seem to work in this case, because the accuracy improves if b is disabled (set to ∞). The e that can improve the accuracy from the baseline ($e = 0$ and $b = \infty$) is $0 < e \leq 2$.

From the experiments, we should prioritize the score of edges computed from a data-driven model rather than weighting the score by DIG for parsing with perfect inputs.

Results of parsing with perfect inputs are shown in Table 2. We use subscript *DIG* to denote the use of DIG while *MAX* represents our learning model i.e. Maximum Entropy Models and MST_{proj} is MSTparser using projective parsing algorithm for training the model.

For overall performance, MAX_{DIG} is the best one. It confirms that the use of the DIG can improve the parsing accuracy of a data-driven parsing model. The accuracy of MST_{proj} is lower than the baseline, even if MSTparser uses the learning model, MIRA [39], that is theoretically better than the model used in the baseline. We think that this is due to the 5-gram prefix features used in MSTParser that does not make sense for Thai, and the simplified part-of-speech features that was added into the baseline model. Also, we see that the better performance of MIRA via the root accuracy of MST_{proj} is higher than the others, because MIRA learns the scores of edges by using a whole dependency tree rather than each pair of dependency nodes.

4.2.3 Parsing with Morphosyntactic Lattices

In this experiment, we created a morphosyntactic lattice by mapping a dictionary using a dynamic backtracking algorithm [40], and generating all the possible word segmentations and part-of-speech tagging results. Here, we assume there is no unknown word in the input text, in order to assure that there is a correct *homophrase*[††] in the lattice. For the

[†]http://sourceforge.net/projects/mstparser

[††]The *homophrase* associated to a trajectory T_p is simply the sequence $H_p = w_{p1} \ldots w_{pl_p}$.

Table 3 Results comparing our systems with the MSTparser where the input is the lattice or the text analyzed by the morphosyntactic analyzer (for MAX^*_{DIG} and $MAX^*_{DIG\text{-}LM}$, we set $k = 10$, $b = 0$ and $e = 5$).

	DP	CR	RA
MAX^d (baseline)	68.01	4.00	75.00
MAX^d_{DIG}	68.39	5.00	77.00
MST^d_{proj}	65.99	4.00	78.00
MAX^*	66.42	4.00	76.00
MAX^*_{DIG}	68.39	4.00	79.00
MAX^*_{LM}	73.43	5.00	79.00
$MAX^*_{DIG\text{-}LM}$	**74.32**	**6.00**	**81.00**

Table 4 Results comparing the accuracy of morphosyntactic analysis.

	TR	TP	TF
trigram model	88.64	87.34	87.99
MAX^*	87.12	88.08	87.59
MAX^*_{DIG}	88.11	88.88	88.49
MAX^*_{LM}	92.86	93.27	93.06
$MAX^*_{DIG\text{-}LM}$	**93.15**	**93.49**	**93.32**

other methods that cannot take the lattice structure as input, we used input texts that were morphosyntactically analyzed by the analyzer[†] instead.

Like in the previous experiment, we also observe the effect of varying the parameters. As we expected, the result is the opposite to experiment with the perfect inputs. The parsing accuracy improved when k increased and was stable where $k \geq 10$ (k varied from 1 to 20). The accuracy is the highest when $b = 0$ and $4 \leq e \leq 5$.

The experiment shows that we can trust the use of DIG more than in the score computed by the learning model, unlike in the previous experiment, because when the input is a lattice, the effect of the independence assumption is more evident, then many invalid dependencies are produced. So, the use of DIG plays an important role in this experiment.

Table 3 compares the results. We use the superscript $*$ to indicate methods taking a lattice as input, and superscript d to indicate that the input is a dubiously disambiguated string. We also used subscript the LM to indicate the use of a language model for rescoring.

The $MAX^*_{DIG\text{-}LM}$ method is the best one, the accuracy is far better that the parsers used on inputs produced by the used morphosyntactical analyzer. Moreover, the accuracy of morphosyntactic analysis of the results is also improved, as shown in Table 4.

The MAX^* method is the worst: the accuracy of parse trees and the accuracy of morphosyntactic analysis are lower than the baseline. That is similar to the MAX^*_{DIG} method, which shows that using only the DIG can slightly improve the parse accuracy only, but does not improve the morphosyntactic analysis. Because the score of morphosyntactic analysis is not taken into account. The use of DIG can improve the parsing accuracy, but it is not enough. By contrast, MAX^*_{LM} uses only the LM but obviously improves the accuracy of morphosyntactic analysis and that of the parse trees.

The results show that if we perform morphosyntactic analysis and syntactic analysis simultaneously by using their information to help each other (here we use the score of morphosyntactic analysis to rescore the parse trees), the accuracy of that combination is better than that of the usual sequence (morphosyntactic analysis followed by syntactic analysis).

Table 5 The accuracy of oracle parse in the 10-best parses.

	DP	CR	RA
MAX^d	72.44	7.00	89.00
MAX^d_{DIG}	73.03	9.00	90.00
MAX^*	74.43	12.00	84.00
MAX^*_{DIG}	**78.02**	**13.00**	**93.00**
MAX^*_{LM}	76.03	11.00	84.00
$MAX^*_{DIG\text{-}LM}$	76.42	11.00	84.00

4.2.4 *K*-Best Parsing

The next experiment is on k-best parsing. We use the same algorithm with the best parameter settings ($b = 0$ and $e = 5$) as in the previous section, and we also study the oracle parse, or the best parse, among the top 10 parses. The result is shown in Table 5. Note that the MSTparser which we used cannot produce k-best parse trees, hence its result is absent.

For the oracle parse, the MAX^*_{DIG} method becomes the best one. In addition, if we increase k to 100, MAX^* also outperforms MAX^*_{LM} and $MAX^*_{DIG\text{-}LM}$. It shows that the models using a LM for rescoring worse in the oracle parse.

We notice that the methods using a LM for rescoring produce a lot of parse trees having duplicated patterns of morphosyntactic analysis. This is because the rescoring by the LM method lets the parsing process consider the score of LM first and the score of dependencies later. In other words, the text of highest score of LM is first selected and as many corresponding parse trees as possible will be generated. It leads to the imbalanced decision making by the parser, that over-emphasizes the score of LM. This problem occurs only in the k-best parsing, it does not affect the best parsing. Therefore, in k-best parsing the rescoring method should not be used.

Figure 10 and 11 show the DP and the TF of the oracle on k-best parsing where k is 10, 20, 30, 50 and 100.

The DP and the TF of oracle in 100-best parsing with the MAX^*_{DIG} method are 81.21% and 95.18%, respectively, and both tend to continuously increase if k increases even more. That is similar to other methods for which the input is a morphosyntactic lattice. Unlike the methods where the input is a dubiously disambiguated string, the DP of oracle tends to improve at first, but slightly increase when k increases even more and becomes saturated when $k \geq 50$: the accuracy of the morphosyntactic analysis cannot improve anymore. In other words, the parsing accuracy is first limited by the accuracy of morphosyntactic analysis of the input. Conversely, if we increase k when inputs are mor-

[†]We used the trigram model trained by the same corpus that was used in the LM rescoring process.

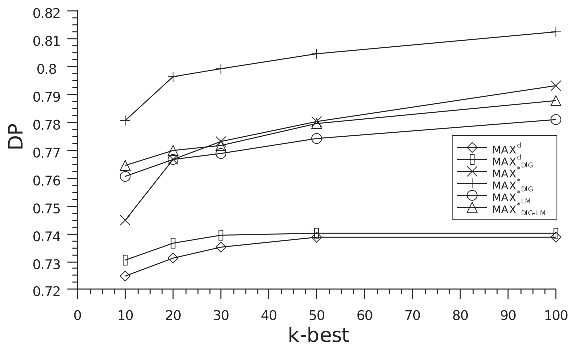


Fig. 10 DP of the *k*-best oracle on the test data.

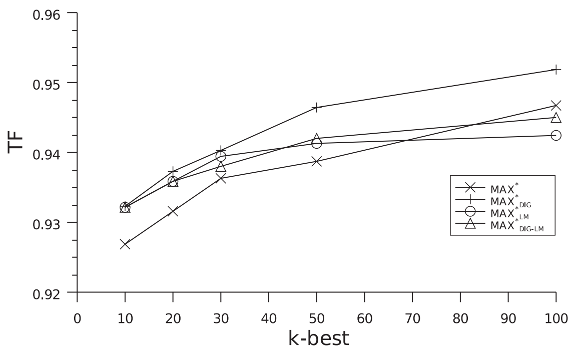


Fig. 11 TF of the *k*-best oracle on the test data.

phosyntactic lattices, the search space for finding both the best morphosyntactic analysis and the best parse tree is enlarged. Hence, the chance of finding the better a best parse tree among *k*-parse trees is higher than when the inputs are dubiously disambiguated strings.

Clearly, the use of a morphosyntactic lattice as input, in case of the *k*-best parsing, significantly and decisively improves the accuracy (relative to the oracle parse) of the morphosyntactic analysis and the parsing process.

5. Conclusion

We have proposed a dependency parsing method for languages without reliable morphosyntactic analysis tools. For this, we represent the input as a morphosyntactic lattice structure and apply an adaptation of Eisner's algorithm to find projective dependency trees. From the experimental results, our method performs better than those made on dubiously disambiguated strings. Moreover, we have also shown how to use Dependency Insertion Grammar (DIG) to adjust the scores computed by the statistical parsing models, and rescore the parse trees by the LM and *k*-best extension of the parser.

The results show that our methods can significantly improve the parsing accuracy. The highest parsing accuracy (DP) reported in this paper is 74.32% which represents 6.31% improvement compared to the model taking the input from the unreliable morphosyntactic analysis tools. Even that accuracy is not enough for high-level NLP applications such as MT, Information Extraction and Text Summarization, it is still useful for corpus preparing process that requests a parser which can produce reasonable *k*-best parse trees in order to let annotators start from the best tree among *k*-best trees rather than from a doubtful parse tree.

There are two points in our parsing model that can be improved in the future. The first one is the combination of morphosyntactic and syntactic process. In this work, we have used the rescoring technique (as mentioned in Sect. 3.2.3). In order to select the best dependency tree more correctly, one might find a better way to seamlessly combine the scores from those two processes together. The second one is utilizing linguistic knowledge for data-driven parsing model. Here, we have used DIG and the adjusting score technique (as mentioned in Sect. 3.2.2). However, it is still worth to try using other dependency grammars that is more sophisticated than DIG.

References

[1] G. Gazdar, E. Klein, G. Pullum, and I. Sag, Generalized Phrase Structure Grammar, Harvard University Press, 1985.
[2] M. Dalrymple, "Lexical functional grammar," Syntax and Semantics, vol.34, 1st ed., Academic Press, 2001.
[3] C. Pollard and I. Sag, Head-Driven Phrase Structure Grammar (Studies in Contemporary Linguistics), University of Chicago Press, 1994.
[4] J. Aravind and Y. Schabes, "Tree-adjoining grammars," Handbook of Formal Languages, vol.3, pp.69–124, 1997.
[5] A. Colmerauer, "Metamorphosis grammars," Natural Language Communication with Computers, Lecture Notes in Computer Science, vol.63, pp.133–189, London, UK, Springer-Verlag, 1978.
[6] F. Pereira and D. Warren, "Definite clause grammars for language analysis," Readings in Natural Language Processing, pp.101–124, 1986.
[7] M. McCord, "Slot grammars," Computational Linguistics, vol.6, no.1, pp.31–43, 1980.
[8] Y. Ding and M. Palmer, "Machine translation using probabilistic synchronous dependency insertion grammars," Proc. 43rd Annual Meeting on Association for Computational Linguistics (ACL-05), pp.541–548, Ann Arbor, Michigan, Association for Computational Linguistics, June 2005.
[9] A. Yakushiji, Y. Miyao, Y. Tateisi, and J. Tsuji, "Biomedical information extraction with predicate-argument structure patterns," Proc. First International Symposium on Semantic Mining in Biomedicine, pp.60–69, Hinxton, Cambridgeshire, UK, 2005.
[10] M. Gagnon and L.D. Sylva, "Text summarization by sentence extraction and syntactic pruning," Proc. 3rd Computational Linguistics in the North East (CliNE-05), Université du Québec en Outaouais, Gatineau, Canada, 2005.
[11] J.J. Kim and J.C. Park, "Annotation of gene products in the literature with gene ontology terms using syntactic dependencies," Proc. 1st International Joint Conference on Natural Language Processing (IJCNLP), pp.528–534, Sanya City, Hainan Island, China, 2004.
[12] B. Vauquois, G. Veillon, and J. Veyrunes, "Application des grammaires formelles aux modèles linguistiques en traduction automatique," Communication au Congrès de Prague, Sept. 1964.
[13] B. Vauquois, G. Veillon, and J. Veyrunes, "Un méatalangage de grammaires transformationnelles: Applications aux problèmes de transfert et de génération syntaxiques," Proc. 1967 Conference on Computational Linguistics, pp.1–50, Stockholm, Association for Computational Linguistics, 1967.
[14] S. Kurohashi and M. Nagao, "KN parser: Japanese dependency/case structure analyzer," Proc. Workshop on Sharable Natural Language

Resources, pp.48–55, Ikoma, Nara, Japan, 1994.

[15] J. Hajič and K. Ribarov, "Rule-based dependencies," Proc. Workshop on the Empirical Learning of Natural Language Processing Tasks, pp.125–136, Prague, Czech Republic, 1997.

[16] U. Germann, "A deterministic dependency parser for Japanese," Proc. MT Summit VII: MT in the Great Translation Era, Singapore, 1999.

[17] K. Ribarov, "On the rule-based parsing of Czech," Prague Bulletin of Mathematical Linguistics, no.77, pp.77–99, 2002.

[18] A. Ratnaparkhi, "Learning to parse natural language with maximum entropy models," Mach. Learn., vol.34, no.1-3, pp.151–175, 1999.

[19] H. Yamada and Y. Matsumoto, "Statistical dependency analysis with support vector machines," Proc. 8th International Workshop on Parsing Technologies, pp.195–206, Nancy, France, 2003.

[20] M. Collins, J. Hajič, E. Brill, L. Ramshaw, and C. Tillmann, "A statistical parser for Czech," Proc. 37th Annual Meeting on Association for Computational Linguistics (ACL-99), pp.505–512, College Park, Maryland, 1999.

[21] M. Collins and N. Duffy, "New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron," Proc. 40th Annual Meeting on Association for Computational Linguistics (ACL-02), pp.263–270, 2002.

[22] K. Uchimoto, S. Sekine, and H. Isahara, "Japanese dependency structure analysis based on maximum entropy models," Proc. 9th Conference on European Chapter of the Association for Computational Linguistics, pp.196–203, Association for Computational Linguistics, Bergen, Norway, 1999.

[23] T. Kudo and Y. Matsumoto, "Japanese dependency structure analysis based on support vector machines," Proc. Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp.18–25, Hong Kong, 2000.

[24] R. McDonald, Discriminative Training and Spanning Tree Algorithms for Dependency Parsing, Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, July 2006.

[25] L. Shen and A.K. Joshi, "Incremental LTAG parsing," Proc. Conference on Human Language Technology and Empirical Methods in Natural Language Processing, pp.811–818, Vancouver, B.C., Canada, 2005.

[26] M. Jinshan, Z. Yu, L. Ting, and L. Sheng, "A statistical dependency parser of Chinese under small training data," Proc. 1st International Joint Conference on Natural Language Processing (IJCNLP), Sanya City, Hainan Island, China, 2004.

[27] R. Hwa, P. Resnik, A. Weinberg, C. Cabezas, and O. Kolak, "Bootstrapping parsers via syntactic projection across parallel texts," Natural Language Engineering, vol. 11, no.3, pp.311–325, 2005.

[28] M. Tomita, "An efficient word lattice parsing algorithm for continuous speech recognition," Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, Tokyo, Japan, 1986.

[29] M. Harper, L. Jamieson, C. Zoltowski, and R. Helzerman, "Semantics and constraint parsing of word graphs," Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, pp.63–66, Minneapolis, Minnesota, 1993.

[30] H. Maruyama, "Structural disambiguation with constraint propagation," Proc. 28th Annual Meeting on Association for Computational Linguistics, pp.31–38, Association for Computational Linguistics, Pittsburgh, Pennsylvania, USA, 1990.

[31] C. Collins, B. Carpenter, and G. Penn, "Head-driven parsing for word lattices," Proc. 42nd Annual Meeting on Association for Computational Linguistics, p.231, Barcelona, Spain, 2004.

[32] M. Collins, Head-driven statistical models for natural language parsing, Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, 1999.

[33] J. Eisner, "Three new probabilistic models for dependency parsing: An exploration," Proc. 16th International Conference on Computational Linguistics (COLING-96), pp.340–345, Copenhagen, Aug. 1996.

[34] Y. Ding and M. Palmer, "Synchronous dependency insertion grammars: A grammar formalism for syntax-based statistical MT," Proc. Workshop on Recent Advances in Dependency Grammars, The 20th International Conference on Computational Linguistics (COLING 2004), Geneva, Switzerland, 2004.

[35] F. Xia, "Extracting tree adjoining grammars from bracketed corpora," 5th Natural Language Processing Pacific Rim Symposium (NLPRS-99), Beijing, China, Nov. 1999.

[36] L. Huang and D. Chiang, "Better k-best parsing," Proc. International Workshop on Parsing Technologies (IWPT), Vancouver, B.C., Canada, 2005.

[37] C. Wacharamanotham, M. Suktarachan, and A. Kawtrakul, "The development of web-based annotation system for Thai," Proc. 7th International Symposium on Natural Language Processing, Pattaya, Chonburi, Thailand, 2007.

[38] L. Shen, Statistical LTAG Parsing, Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, 2006.

[39] K. Crammer, O. Dekel, J. Keshat, and S. Shalev-Shwartz, "Online passive-aggressive algorithms," Journal of Machine Learning Research, vol. 7, pp. 551–585, March 2006.

[40] C. Thumkanon, "A statistical model for Thai morphological analysis," Master's Thesis, Computer Engineering, Kasetsart University, 2001.

Appendix A: Abbreviation of Thai Part-of-Speeches and Grammatical Functions

A.1 Part-of-Speeches

1. npn (proper noun)
2. nnum (cardinal number)
3. norm (ordinal number marker)
4. nlab (label noun)
5. ncn (common noun)
6. nct (collective noun)
7. ntit (title noun)
8. pper (personal pronoun)
9. pdem (demonstrative pronoun)
10. pind (indefinite pronoun)
11. ppos (possessive pronoun)
12. prfx (reflexive pronoun)
13. prec (reciprocal pronoun)
14. prel (relative pronoun)
15. pint (interrogative pronoun)
16. vi (intransitive verb)
17. vt (transitive verb)
18. vcau (causative verb)
19. vcs (complementary state verb)
20. vex (existential verb)
21. prev (pre-verb)
22. vpost (post-verb)
23. honm (honorific marker)
24. det (determiner)
25. indet (indefinite determiner)
26. adj (adjective)
27. adv (adverb)
28. advm1 (adverb marker1)
29. advm2 (adverb marker2)
30. advm3 (adverb marker3)

31. advm4 (adverb marker4)
32. advm5 (adverb marker5)
33. cl (classifier)
34. conj (conjunction)
35. conjd (double conjunction)
36. conjncl (noun clause conjunction)
37. prep (preposition)
38. prepc (co-preposition)
39. int (interjection)
40. pref1 (prefix1)
41. pref2 (prefix2)
42. pref3 (prefix3)
43. aff (affirmative)
44. part (particle)
45. neg (negative)
46. punc (punctuation)
47. idm (idiom)
48. psm (passive voice marker)
49. sym (symbol)

A.2 Grammatical Functions

Complements

1. subject (subj)
2. clausal subject (csubj)
3. direct object (dobj)
4. indirect object (iobj)
5. prepositional object (pobj)
6. prepositional complement (pcomp)
7. subject or object predicative (pred)
8. clausal predicative (cpred)
9. conjunction (conj)
10. subordinating conjunction (sconj)
11. nominalizer (nom)
12. adverbalizer (advm)

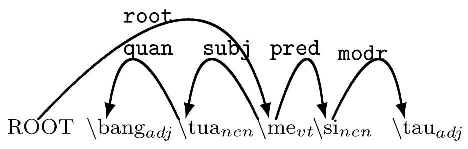
Adjuncts

1. parenthetical modifier (modp)
2. restrictive modifier (modr)
3. mood modifier (modm)
4. aspect modifier (moda)
5. locative modifier (modl)
6. parenthetical apposition (appa)
7. restrictive apposition (appr)
8. relative clause modification (rel)
9. determiner (det)
10. quantifier (quan)
11. classifier (cl)
12. coordination (coord)
13. negation (neg)
14. punctuation (punc)
15. double preposition (dprep)
16. parallel serial verb (svp)
17. sequence serial verb (svs)

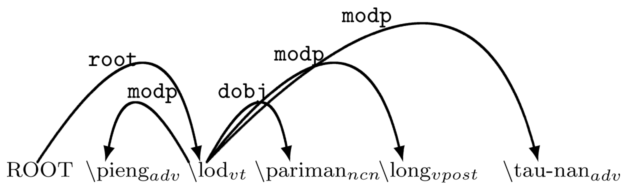
Appendix B: Examples of NAiST Dependency Treebank, Extracted Elementary Trees, and *k*-Best Parse Trees

B.1 NAiST Dependency Treebank

1. \bang(some) \tua(one) \me(is) \si(color) \tau(gray)

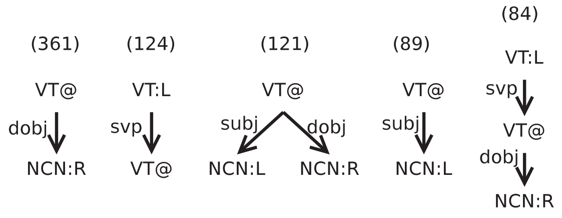


2. \pieng(just) \lod(decrease) \pariman(quantity) \long(down) \tau-nan(only)



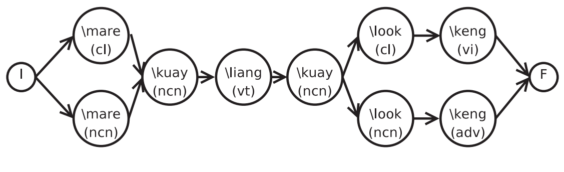
B.2 Extracted Elementary Trees

Top 5 of the most occurrence relaxed elementary trees of transitive verb (vt)



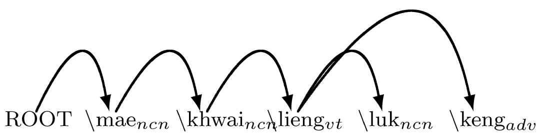
B.3 *K*-Best Parse Trees

Input: “\mare(mother) \kuay(buffalo) \liang(take care) \look(kid) \keng(well)”

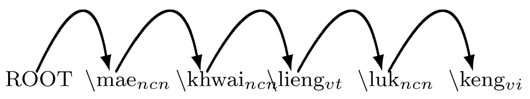


Output: 5-best unlabeled parse trees

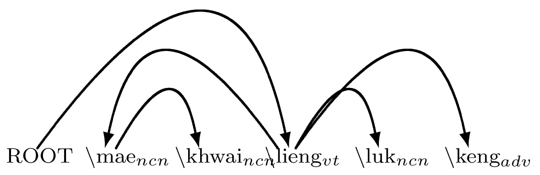
1. score=2.863449



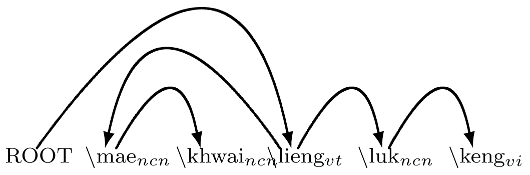
2. score=2.760695


ROOT $\backslash$mae$_{ncn}$ $\backslash$khwai$_{ncn}$ $\backslash$lieng$_{vt}$ $\backslash$luk$_{ncn}$ $\backslash$keng$_{vi}$

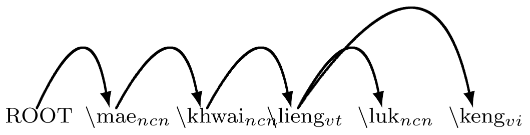
3. score=-0.622163 (correct tree)


ROOT $\backslash$mae$_{ncn}$ $\backslash$khwai$_{ncn}$ $\backslash$lieng$_{vt}$ $\backslash$luk$_{ncn}$ $\backslash$keng$_{adv}$

4. score=-0.724917


ROOT $\backslash$mae$_{ncn}$ $\backslash$khwai$_{ncn}$ $\backslash$lieng$_{vt}$ $\backslash$luk$_{ncn}$ $\backslash$keng$_{vi}$

5. score=-0.840073


ROOT $\backslash$mae$_{ncn}$ $\backslash$khwai$_{ncn}$ $\backslash$lieng$_{vt}$ $\backslash$luk$_{ncn}$ $\backslash$keng$_{vi}$

Sutee Sudprasert received the Bachelor's degree and Master's degree in computer engineering from Kasetsart University, Bangkok, Thailand. He is now a candidate for the Doctoral program of Computer Engineering at Kasetsart University. His general research interest in natural language processing, especially morphosyntactic analysis, syntactic parser, and machine translation.

Asanee Kawtrakul received the Bachelor's degree and Master's degree in electrical engineering from Kasetsart University, Bangkok, Thailand, Doctoral degree in information engineer from Nagoya University, Japan. Currently, she is an associate professor of computer engineering at Kasetsert University and deputy director of National Electronics and Computer Technology Center. Her research interests include natural language processing and knowledge engineering.

Christian Boitet is the director of GETA, the Groupe d'\'Etude pour la Traduction Automatique, long a world leader in the field of machine translation (MT), and codirector of GETALP (Study Group on Machine Translation and Processing of Languages and Speech), a larger group formed in 2007 by joining with GEOD, a group researching speech and dialogue processing. On graduating from the \'Ecole Polytechnique (Paris) in 1970, he joined the Centre Nationale de la Recherche Scientifique (CNRS) and Prof. Bernard Vauquois' pioneering MT team (GETA) at the Université Joseph Fourier in Grenoble. In parallel, he pursued studies in various languages and obtained a diploma in Russian from the Institut National des Langues et Civilizations Orientales in 1973. He presented his State Doctoral Thesis (on several mathematical and algorithmic problems related to MT) in 1976. In 1977, he became Associate Professor of Computer Science, teaching algorithms, compiler construction, formal languages and automata, elementary logic, formal systems, and natural language processing. He became full professor in 1987. His current research interests include multilingual interpersonal communication over networks (project UNL); machine-assisted human translation; speech translation (project CSTAR II); personal dialog-based MT for monolingual authors; portable and readable encodings for multilingual documents; computational tools for revisers and occasional translators; multilingual lexical data bases; and specialized languages and environments for linguistic engineering and research.

Vincent Berment received two B.S. in Mathematics from Aix-Marseille University and in Computer Science from Paris University and also received two B.A. in Laotian language and Contemporary Asian Civilization from the National Institute of the Oriental Civilizations and Languages (INALCO), Paris. He received M.S. in Electrical Engineering from Supélec, M.BA. in Business Administration from Sorbonne and M.A. in Computational linguistics from INALCO. In 2004, he received the PhD. in Computer Science from Grenoble University. From 1998-2000, he was a lecturer at INALCO, teaching in Lao language processing and Lao grammar. Now he works at CS Syst?mes d'Information company and also works as a associate researcher at the Machine Translation Group (GETA), Grenoble.