Efficient retrieval of tree translation examples for Syntax-Based Machine Translation

Fabien Cromieres
Graduate School of Informatics
Kyoto University
Kyoto, Japan
fabien@nlp.kuee.kyoto-u.ac.jp

Sadao Kurohashi
Graduate School of Informatics
Kyoto University
Kyoto, Japan
kuro@i.kyoto-u.ac.jp

Abstract

We propose an algorithm allowing to efficiently retrieve example treelets in a parsed tree database in order to allow on-the-fly extraction of syntactic translation rules. We also propose improvements of this algorithm allowing several kinds of flexible matchings.

1 Introduction

The popular Example-Based (EBMT) and Statistical Machine Translation (SMT) paradigms make use of the translation examples provided by a parallel bilingual corpus to produce new translations. Most of these translation systems process the example data in a similar way: The parallel sentences are first word-aligned. Then, translation rules are extracted from these aligned sentences. Finally, the translation rules are used in a decoding step to translate sentences. We use the term translation rule in a very broad sense here, as it may refer to substring pairs as in (Koehn et al., 2003), synchronous grammar rules as in (Chiang, 2007) or treelet pairs as in (Quirk et al., 2005; Nakazawa and Kurohashi, 2008).

As the size of bilingual corpus grow larger, the number of translation rules to be stored can easily become unmanageable. As a solution to this problem in the context of phrase-based Machine Translation, (Callison-Burch et al., 2005) proposed to pre-align the example corpora, but delay the rule extraction to the decoding stage. They showed that using Suffix Arrays, it was possible to efficiently retrieve all sentences containing substrings of the sentence to be translated, and thus extract the needed translation rules on-the-fly. (Lopez, 2007) proposed an extension of this method for retrieving discontinuous substrings, making it suitable for systems such as (Chiang, 2007).

In this paper, we propose a method to apply the same idea to Syntax-Based SMT and EBMT (Quirk et al., 2005; Mi et al., 2008; Nakazawa and Kurohashi, 2008). Since Syntax-Based systems usually work with the parse trees of the source-side sentences, we will need to be able to retrieve efficiently examples trees from fragments (treelets) of the parse tree of the sentence we want to translate. We will also propose extensions of this method allowing more flexible matchings.

2 Overview of the method

2.1 Treelet retrieval

We first formalize the setting of this chapter by providing some definitions.

Definition 2.1 (Treelets). A *treelet* is a connected subgraph of a tree. A treelet T_1 is a *subtreelet* of another treelet T_2 if T_1 is itself a connected subgraph of T_2. We note $|T|$ the number of nodes in a treelet. If $|T| = 1$, T is called an *elementary treelet*. A *linear treelet* is a treelet whose nodes have at most 1 child. A *subtree* rooted at node n of a tree $\mathcal{T}$ is a treelet containing all nodes descendants of n.

Definition 2.2 (Sub- and Supertreelets). If T_1 is a subtreelet of T_2 and $|T_1| = |T_2| - 1$, we call T_1 an *immediate subtreelet* of T_2. Reciprocally, T_2 is an *(immediate) supertreelet* of T_1. Furthermore, if T_2 and T_1 are rooted at the same node in the original tree, we say that T_2 is a *descending supertreelet* of T_1. Otherwise it is an *ascending supertreelet* of T_1.

Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, pages 508–518,
Edinburgh, Scotland, UK, July 27–31, 2011.

In treelet retrieval, we are given a certain treelet type and want to find all of the tokens of this type in the database $\mathcal{DB}$. Each token of a given treelet type will be identified by a mapping from the node of the treelet type to the nodes of the treelet token in the database.

Definition 2.3 (Matching). Given a treelet T and a tree database $\mathcal{DB}$, a matching of T in $\mathcal{DB}$ is a function M that associate the treelet T to a tree $\mathcal{T}$ in $\mathcal{DB}$ and every node of T to nodes of $\mathcal{T}$ in such a way that: $\forall n \in T, label(M(n)) = label(n)$ and $\forall (n_1, n_2) \in T$ s.t n_2 is a child of n_1, $M(n_2)$ is a child of $M(n_1)$.

In the common case where the siblings of a tree are ordered, a matching must satisfy the additional restriction: $\forall n_1, n_2 \in T, n_1 <_s n_1 \Leftrightarrow M(n_1) <_s M(n_1)$, where $<_s$ is the partial order relation between nodes meaning "is a sibling and to the left of"

We note $occ(T)$ (for "occurrences of T ") the set of all possible matchings from T to $\mathcal{DB}$. We will call *computing* T the task of finding $occ(T)$. If $|occ(T)| = 0$, we call T an *empty treelet*. *Computing a query tree* $\mathcal{T}_Q$ means computing all of its treelets.

Definition 2.4 (Notations). Although treelets are themselves trees, we will use the word *treelet* to emphasize they are a subpart of a bigger tree. We will note T a treelet, and $\mathcal{T}$ a tree. $\mathcal{T}_Q$ is the query tree we want to compute. $\mathcal{DB}$ will refer to the set of trees in our database. We will use a bracket notation to describe trees or treelets. Thus "a(b c d(e))" is the tree at the bottom of figure 2.

2.2 General approach

There exists already a large body of research about tree pattern matching (Dubiner et al., 1994; Bruno et al., 2002). However, our problem is quite different from finding the tokens of a given treelet in a database. We actually want to find all the tokens of all of the treelets of a given query tree. The query tree itself is unlikely to appear in full even once in the database. In this respect, our approach will have many similarities with (Callison-Burch et al., 2005) and (Lopez, 2007), and can be seen as an extension of these works.

The basis of the method in (Lopez, 2007) is to look for the occurrences of continuous substrings using a Suffix Array, and then intersect them to find the occurrences of discontinuous substrings. We will have a similar approach with two variants. The first variant consists in using an adaptation of the concept of suffix arrays to trees, which we will call Path-To-Root Arrays (section 3.4), that allows us to find efficiently the set of occurrences of a linear treelet. Occurrences of non-linear treelets can then be computed by intersection. The second variant is to use an inverted index (section 3.5). Then the occurrences of all treelets, even the linear treelets, are computed by intersection.

The main additional difficulty in considering trees instead of strings is that while a string has a quadratic number of continuous substrings, a tree has in general an exponential number of treelets (eg. several trillion for the dependency tree of a 70 words sentence). There is also an exponential number of discontinuous substrings, but (Lopez, 2007) only consider substrings of bounded size, limiting this problem. We will not try to bound the size of treelets retrieved. It is therefore crucial to avoid computing the occurrences of treelets that have no occurrences in the database, and also to eliminate as much redundant calculation as is possible.

Lopez proposes to use Prefix Trees for avoiding any redundant or useless computation. We will use a similar idea but with an hypergraph that we will call "computation hypergraph" (section 3.2). This hypergraph will not only fit the same role as the Prefix Tree of (Lopez, 2007), but also will allow us to easily implement different search strategies for flexible search (section 6).

2.3 Representing positions

Whether we use a Path-to-Root Array or an inverted index, we will need a compact way to represent the position of a node in a tree. It is straightforward to define such a position for strings, but slightly less for trees. Especially, if we consider ordered trees, we will want to be able to compare the relative location of the nodes by comparing their positions.

The simplest possibility is to use an integer corresponding to the rank of the node in a in-order depth-first traversal of the tree. It is then easy, for two nodes b and c, children of a parent node a, to check if b is on the left of c, or on the left of a, for example.

A more advanced possibility is to use a representation inspired from (Zhang et al., 2001), in which

the position of a node is a tuple consisting of its rank in a preorder (ie. children last) and a postorder (children first) depth-first traversal, and of its distance to the root. This allows to test easily whether a node is an ancestor of another, and their distance to each other. This allows in turn to compute by intersection the occurrences of discontinuous treelets, much like what is done in (Lopez, 2007) for discontinuous strings. This is discussed in section 7.2.

3 Computing treelets incrementally

We describe here in more details how the treelets can be efficiently computed incrementally.

3.1 Dependence of treelet computation

Let us first define how it is possible to compute a treelet from two of its subtreelets. Let us consider a treelet T and two treelets T_1 and T_2 such that $T = T_1 \cup T_2$, where, in the equality and the union, the treelet are seen as the set of their nodes. There are two possibilities. If $T_1 \cap T_2 = \emptyset$, then the root of T_1 is a child of a node of T_2 or vice-versa. We then say that T_1 and T_2 form a disjoint coverage (abbreviated as D-coverage) of T. If $T_1 \cap T_2 \neq \emptyset$, we will say that T_1 and T_2 form an overlapping coverage (abbreviated as O-coverage) of T.

Given two treelets T_1 and T_2 forming a coverage of T, we can compute $occ(T)$ from $occ(T_1)$ and $occ(T_2)$ by combining their matchings.

Definition 3.1 (compatibility for O-coverage). Let T be a treelet of $\mathcal{T}_Q$. Let T_1 and T_2 be 2 treelets forming a O-coverage of T. Let $M_1 \in occ(T_1)$ and $M_2 \in occ(T_2)$. M_1 and M_2 are compatible if and only if $M_1|_{T_1 \cap T_2} = M_2|_{T_1 \cap T_2}$ and $\mathcal{I}(M_1|_{T_1 \setminus T_2}) \cap \mathcal{I}(M_2|_{T_2 \setminus T_1}) = \emptyset$.

In the definition above, $|_S$ is the restriction of a function to a set S and $\mathcal{I}$ is the image set of a function.

If the children of a tree are ordered, we must add the additional restriction: $\forall (n_1, n_2) \in (T_1 \setminus T_2) \times (T_2 \setminus T_1), n_1 <_s n_2 \Leftrightarrow M_1(n_1) <_s M_2(n_2)$.

Definition 3.2 (compatibility for D-coverage). Let T_1 and T_2 be 2 treelets forming a D-coverage of T. Let's suppose that the root n_2 of T_2 is a child of node n_1 of T_1. Let $M_1 \in occ(T_1)$ and $M_2 \in occ(T_2)$. M_1 and M_2 are compatible if and only if $M_2(n_2)$ is a child of $M_1(n_1)$.

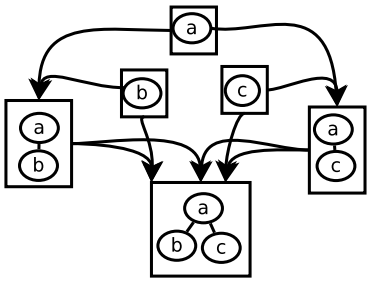


Figure 1: A computing hypergraph for "a(b c)".

Definition 3.3 (intersection ($\otimes$) operation). If two matchings are compatible, we can form their union, which is defined as $(M_1 \cup M_2)(n) = M_1(n)$ if $n \in T_1$ and $M_2(n)$ else. We note $occ(T_1) \otimes occ(T_2) = \{M_1 \cup M_2 \mid M_1 \in occ(T_1), M_2 \in occ(T_2)$ and M_1 is compatible with $M_2\ \}$. Then,we have the property: $occ(T) = occ(T_1) \otimes occ(T_2)$

In practice, the intersection operation will be implemented using merge and binary merge algorithms (Baeza-Yates and Salinger, 2005), following (Lopez, 2007).

3.2 The computation hypergraph

We have seen that it is possible to compute $occ(T)$ from two subtreelets forming a coverage of T. This can be represented by a hypergraph in which nodes are all the treelets of a given query tree, and every pair of overlapping or adjacent treelet is linked by an hyperedge to their union treelet. Whenever we have computed two starting points of an hyper-edge, we can compute its destination treelet. An example of a small computation hypergraph is described in figure 1.

It is very convenient to represent the incremental computation of the treelets as a traversal of this hypergraph. First because it contributes to avoid redundant computations: each treelet is computed only once, even if it is used to compute several other treelets. Also, if a query tree contains two distinct but identical treelets, only one computation will be done, provided the two treelets are represented by the same node in the hypergraph. The hypergraph also allows us to avoid computing empty treelets, as we describe in next section. This hypergraph therefore has the same role for us as the prefix tree used

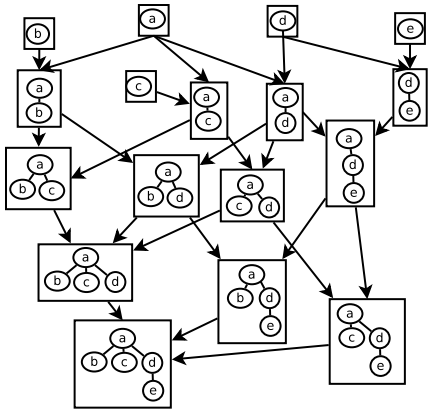


Figure 2: Inclusion DAG for the tree $a(bcd(e))$

in (Lopez, 2007). Of course, the hypergraph is generated on-the-fly during the traversal.

Furthermore, different traversals will define different computation strategies, and we will be able to use some more advanced graph exploration methods in section 6.

3.3 The Immediate Inclusion DAG

In many cases (but not always: see section 4.3), the most optimal computation strategy should be to always compute a treelet from two of its immediate subtreelets. This is because the computation time will be proportional to the size of the smallest occurrence set of the two treelets, and thus the "cheapest" subtreelet is always one of the immediate subtreelets. With this computation strategy, we can replace the general computation hypergraph by a DAG (Directed Acyclic Graph) in which every treelet point to its immediate supertreelets. An example is given on figure 2. We will call this DAG the (Immediate) Inclusion DAG.

Traversals of the Inclusion DAG should be pruned when an empty treelet is found, since all of its supertreelets will also be empty. The algorithm 1 provide a general traversal of the DAG avoiding to compute as many empty treelets as possible. It uses a queue $\mathcal{D}$ of discovered treelets, and a data-structure $\mathcal{C}$ that associate a treelet to those of its subtreelets that have been already computed. Once a treelet T has been computed and is found to be non empty, we discover its immediate supertreelets $T^{S1}, T^{S2}, \ldots$ (if they have not been discovered already) and add T to $\mathcal{C}$ (T^{S1}), $\mathcal{C}$ (T^{S2}), ... The operation min($\mathcal{C}$ (T)) retrieve the 2 subtreelets from $\mathcal{C}$ (T) that have the least occurrences. If one of them is empty, we can directly conclude that T is empty. No treelet whose all immediate subtreelets are empty is ever put in the discovered queue, which allows us to prune most of the empty treelets of the Inclusion DAG.

Algorithm 1: Generic DAG traversal

```
1  Add the set of precomputed treelets to 𝒟;
2  while ∃T ∈ 𝒟 s.t T ∈ precomputed or |𝒞(T)| > 2
   do
3      pop T from 𝒟;
4      if T in precomputed then
5          occ(T) ← precomputed[T];
6      else
7          T1,T2=min(𝒞 (T));
8          if |occ(T1)| = 0 then
9              occ(T) ← ∅;
10         else
11             occ(T) ← occ(T1) ⊗ occ(T2);
12     for T^S ∈ supertree(T) do
13         if occ(T^S) = undef then
14             Add T to 𝒞(T^S);
15             if |occ(T)| > 0 and T^S ∉ 𝒟 then
16                 Add T^S to 𝒟;
```

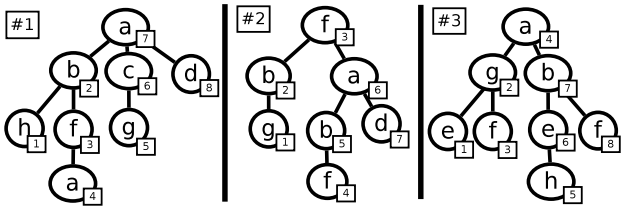
A treelet in the inclusion DAG can be computed as soon as two of its antecedents have been computed. To start the computation (or rather, "seed" it), it is necessary to know the occurrences of treelet of smaller size. In the following sections 3.4 and 3.5, we describe two methods for efficiently obtaining the set of occurrences of some initial treelets.

3.4 Path-to-Root Array

We present here a method to compute very efficiently $occ(T)$ when T is linear. This method is similar to the use of Suffix Arrays (Manber and Myers, 1990) to find the occurrences of continuous substrings in a text.

Definition 3.4 (Paths-to-Root Array). Given a labeled tree $\mathcal{T}$ and a node $n \in \mathcal{T}$, the path-to-root of n is the sequence of labels from n to the root. The *Paths-to-Root Array* of a set of trees $\mathcal{DB}$ is the lexicographically sorted list of the Path-to-Roots of every node in $\mathcal{DB}$.

Just as with suffixes, a path-to-root can be represented compactly by a pointer to its starting node in $\mathcal{DB}$. We then need to keep the database $\mathcal{DB}$ in



	Pos	PtR		Pos	PtR		Pos	PtR
1	3:4	a	8	2:2	bf	15	1:3	fba
2	1:7	a	9	1:6	ca	16	3:3	fga
3	2:6	af	10	1:8	da	17	3:2	ga
4	1:4	afba	11	2:7	daf	18	2:1	gbf
5	3:7	ba	12	3:1	ega	19	1:5	gca
6	1:2	ba	13	2:3	f	20	1:1	hba
7	2:5	baf	14	3:8	fba	21	3:5	heba

Figure 3: Path To Root Array for a set of three trees. "Pos." is the position of the starting point of a given path-to-root (noted as indexOfTree:positionInTree), and PtR is the sequence of labels on this path-to-root. The path-to-root are sorted in lexicographic order. We can find the set of occurrences of any linear treelet with a binary search. For example, the treelet a(b) corresponds to the label sequence "ba". With a binary search, we find that the path-to-root starting with "ba" are between indexes 5 and 7. The corresponding occurrences are then 3:7, 1:2 and 2:5.

memory to retrieve efficiently the pointed-to path-to-root. Once the Path-to-Root Array is built, for a linear treelet T, we can find its occurrences by a binary search of the first and last path-to-root starting with the labels of T. See figure 3 for an example.

Memory cost is quite manageable, since we only need 10 bytes per nodes in total. 5 bytes per pointer in the array (tree id: 4 bytes, start position: 1 byte), and 5 bytes per nodes to store the database in memory (label id:4 bytes, parent position: 1 byte).

All the optimization tricks proposed in (Lopez, 2007) for Suffix Arrays can be used here, especially the optimization proposed in (Zhang and Vogel, 2005).

3.5 Inverted Index and Precomputation

Instead of a Path-to-Root array, one can simply use an inverted index. The inverted index associates with every label the set of its occurrences, each occurrences being represented by a tuple containing the index of the tree, the position of the label in the tree, and the position of the parent of the label in the tree. Knowing the position of the parent will allow to compute treelets of size 2 by intersection (D-coverage). This is less effective than the Path-To-Root Array approach, but open the possibilities for the flexible search discussed in section 6.

Taking the idea further, we can actually consider the possibility of precomputing treelets of size greater than 1, especially if they appear frequently in the corpus.

4 Practical implementation of the traversal

4.1 Postorder traversal

The way we choose the treelet to be popped out on line 3 of algorithm 1 will define different computation strategies. For concreteness, we describe now a more specific traversal. We will process treelets in an order depending on their root node. More precisely, we consider the nodes of the query tree in the order given by a depth-first postorder traversal of the query tree. This way, when a treelet rooted at n is processed, all of the treelets rooted at a descendant of n have already been processed.

We can suppose that every processed treelet is assigned an index that we note $\#T$. This allows a convenient recursive representation of treelets.

Definition 4.1 (Recursive representation). Let T be a treelet rooted at node n of $\mathcal{T}_Q$. We note n_i the i^{th} child of n in $\mathcal{T}_Q$. For all i, t_i is the subtree of T rooted at n_i. We note $t_i = \emptyset$ and $\#t_i = 0$ if T does not contain n_i. The recursive representation of T is then: $[n, (\#t_1, \#t_2, \ldots, \#t_m)]$. We note T^i the value $\#t_i$.

For example, if $\mathcal{T}_Q$ ="a(b c d(e))" and the treelets "b" and "d(e)" have been assigned the indexes 2 and 4, the recursive representation of the treelet "a(b d(e))" would be [a,(2,0,4)].

Algorithm 2 describes this "postorder traversal". $\mathcal{D}_{Node}$ is a priority queue containing the treelets rooted at $Node$ discovered so far. The priority queue pop out the smallest treelets first. Line 14 maintain a list $\mathcal{L}$ of processed treelets and assign the index of T in $\mathcal{L}$ to $\#T$. Line 22 keeps track of the non-empty immediate supertreelets of every treelet through a dictionary $\mathcal{S}$. This is used in the procedure *compute-supertreelets* (algorithm 3) to generate the immediate supertreelets of a treelet T given its recursive representation. In this procedure, line 6 produces the

Algorithm 2: DAG traversal by query-tree postorder

```
1  for Node in postorder-traversal(query-tree) do
2      T_elem = [Node, (0, 0, .., 0)];
3      D_Node ← T_elem;
4      while |D_Node| > 0 do
5          T=pop-first(D_Node);
6          if T in precomputed then
7              occ(T) ← precomputed[Node.label];
8          else
9              T_1, T_2=min(C (t));
10             if |occ(T_1)| = 0 then
11                 occ(T) ← ∅;
12             else
13                 occ(T) ← occ(T_1) ⊗ occ(T_2);
14         Append T to L;
15         #T ← |L|;
16         for T^S in compute-supertree(T, #T) do
17             Add T to C(T^S);
18             if |occ(T)| > 0 then
19                 if T^S ∉ D_Node and
                   root(T^S)=Node then
20                     Add T^S to D;
21                 for #t in C (T) do
22                     Add #T to S(#t);
```

Algorithm 3: compute-supertrees

```
   Input: T,#T
   Output: lst: list of immediate supertreelets of T
1  m ← |root(T)|;
2  for i in 1 . . . m do
3      for #T^S in S(#T^i) do
4          if root(#T^S) ≠ root(T) then
5              T_new ← [root(T), T^0, ..#T', . . . , T^m];
6              Append T_new to lst;
7  T_new ← [parent(root(T)), (0, . . . , #T, . . . , 0)];
8  Append T_new to lst;
```

descending supertreelets, and line 8 produces the ascending supertreelet. Figure 4 describes the content of all these data structures for a simple run of the algorithm.

This postorder traversal has several advantages. A treelet is only processed once all of its immediate supertreelets have been computed, which is optimal to reduce the cost of the $\otimes$ operation. The way the procedure *compute-supertreelets* discover supertreelets from the info in $\mathcal{S}$ has also several benefit. One is that, by not adding empty treelets (line 18) to $\mathcal{S}$, we naturally prevent the discovery of larger empty treelets. Similarly, in the next section, we will be able to prevent the discovery of non-maximal treelets by modifying $\mathcal{S}$. Modifications of *compute-supertreelets* will also allow different kind of retrieval in section 6.

4.2 Pruning non-maximal treelets

We now try to address another aspect of the overwhelming number of potential treelets in a query tree. As we said, in most practical cases, most of the larger treelets in a query tree will be empty. Still, it is possible that some tree exactly identical to the query tree (or some tree having a very large treelet in common with the query tree) do exist in the database. This case is obviously a best case for translation, but unfortunately could be a worst-case for our algorithm, as it means that all of the (possibly trillions of) treelets of the query tree will be computed.

To solve this issue, we try to consider a concept analogous to that of maximal substring, or substring class, found in Suffix Trees and Suffix Arrays (Yamamoto and Church, 2001). The idea is that in most cases where a query tree is "full" (that is all of its treelets are not empty), most of the larger treelets will share the same occurrences (in the database trees that are very similar to the query tree). We formalize this as follow:

Definition 4.2 (domination and maximal treelets). Let T_1 be a subtreelet of T_2. If for every matching M_1 of $occ(T_1)$, there exist a matching M_2 of $occ(T_2)$ such that $M_2|_{T1} = M_1$, we say that T_1 is dominated by T_2. A treelet is maximal if it is not dominated by any other treelet.

If T_1 is dominated by T_2, it means that all occurrences of T_1 are actually part of an occurrence of T_2. We will therefore be, in general, more interested by the larger treelet T_2 and can prune as many non-maximal treelets as we want in the traversal. The key point is that the algorithm has to avoid discovering most non maximal treelets.The algorithm 2 can easily be modified to do this. We will use the following property.

Property 4.1. Given k treelets $T_1 \dots T_k$ with k distinct roots, all the roots being children of a same node n. We note $n(T_1 \dots T_k)$ the treelet whose root is n, and for which the k subtrees rooted at the k

T	d	e	b	b(d) *[Empty]*	b(e)	b(d e) *[Empty]*	c	a	a(b)	a(b(e))	a(c)	a(b c)	a(b(e) c)
#	1	2	3	4	5	6	7	8	9	10	11	12	13
R	d	e	b(..)	b(1.)	b(.2)	b(1 2)	c	a(..)	a(3.)	a(5.)	a(.7)	a(3 7)	a(5 7)
$\mathcal{C}$	-	-	-	1,3	2,3	4,5	-	-	8,3	5,9	7,8	9,11	10,12
$\mathcal{S}$	-	-	5	-	-	-	-	9,11	10,12	13	12	13	-

Figure 4: A run of the algorithm 2, for the query tree a(b(d e) c). The row "T" represents the treelets in the order they are discovered. The row "#" is the index #T, and the row "R" is the recursive representation of the treelet. Also represented are the content of $\mathcal{C}$ and $\mathcal{S}$ at the end of the computation. When a treelet is poped out of $\mathcal{D}_{Node}$, occ(T) is computed from the treelets listed in $\mathcal{C}$ (T). If occ(T) is not empty, the entries of the immediate subtreelets of T in $\mathcal{S}$ are updated with #T. We suppose here that $|occ($ b(d))|=0. Then, b(d e) is marked as empty and neither b(d) nor b(d e) are added to the entries of their subtreelets in $\mathcal{S}$. This way, when considering treelets rooted at the upper node "a", the algorithm will not discover any of the treelets containing b(d).

children of n are $T_1 \dots T_k$. Let us further suppose that for all i, T_i is dominated by a descending supertreelet T_i^d (with the possibility that $T_i = T_i^d$). Then $n(T_1 \dots T_k)$ is dominated by $n(T_1^d \dots T_k^d)$. For example, if b(c) is dominated by b(c d), then a(b(c) e) will be dominated by a(b(c d) e).

In algorithm 2, after processing each node, we proceed to a cleaning of the $\mathcal{S}$ dictionary in the following way: for every treelet T (considering the treelets by increasing size) that is dominated by one of its supertreelets $T^S \in \mathcal{S}(T)$ and for every subtreelet T' of T such that $T \in \mathcal{S}(T')$, we replace T by T^S in $\mathcal{S}(T')$. The procedure *compute-supertreelets*, when called during the processing of the parent node, will thus skip all of the treelets that are "trivially" dominated according to property 4.1.

Let's note that testing for the domination of a treelet T by one of its supertrelets T^S is not a matter of just testing if $|occ(T)| = |occ(T^S)|$, as would be the case with substring: a treelet can have less occurrences than one of its supertreelets (eg. b(a) has more occurrences than b in b(a a)). An efficient way is to first check that the two treelets occurs in the same number of sentences, then confirm this with a systematic check of the definition.

4.3 The case of constituent trees

We have focused our experiments on dependency trees, but the method can be applied to any tree. However, the computations strategies we have used might not be optimal for all kind of trees. In a dependency tree, nodes are labeled by words and most non-elementary treelets have a small number of occurrences. In a constituent tree, many treelets containing only internal nodes have a high frequency and will be expensive to compute.

If we have enough memory, we can solve this by precomputing the most common (and therefore expensive) treelets.

However, it is usually not very interesting to retrieve all the occurrences of treelets such as "NP(Det NN)" in the context of a MT system. Such very common pattern are best treated by some pre-computed rules. What is interesting is the retrieval of lexicalized rules. More precisely, we want to retrieve efficiently treelets containing at least one leaf of the query tree. Therefore, an alternative computation strategy would only explore treelets containing at least one terminal node. We would thus compute successively "dog", "NN(dog)" "NP(NN(dog))", "NP(Det NN(dog))", etc.

4.4 Complexity

Processing time will be mainly dependent on two factors: the number of treelets in a query tree that need to be computed, and the average time to compute a treelet.

Let N_C be the size of the corpus. It can be shown quite easily that the time needed to compute a treelet with our method is proportional to its number of occurrences, which is itself growing as $O(N_C)$.

Let m be the size of the query tree. The number of treelets needing to be computed is, in the worst case, exponential in m. In practice, the only case where most of the treelets are non-empty is when the database contains trees similar to the query tree in the database, and this is handled by the modification of the algorithm is section 4.2. In other cases, most of the treelets are empty, and empirically, we find that the number of non-empty treelets in a query tree

Database size (#nodes)	6M	60M
Largest non-empty treelet size	4.6	8.7
Processing time (PtR Array)	0.02 s	0.7 s
Processing time (Inv. Index)	0.02 s	0.9 s
Size on disk	40 MB	500 MB

Figure 5: Performances averaged on 100 sentences.

Method	Treelet dictionary	Our method
Disk space used	23 GB	500 MB
BLEU	11.6%	12.0%

Figure 6: Comparison with a dictionary-based baseline (performances averaged over 100 sentences).

grows approximately as $O(m \cdot N_C^{0.5})$. It is also possible to bound the size of the retrieved treelets (only retrieving treelets with less than 10 nodes, for example), similarly to what is done in (Lopez, 2007). The number of treelets will then only grows as $O(m)$.

The total processing time of a given query tree will therefore be on the order of $O(m \cdot N_C^{1.5})$ (or $O(m \cdot N_C)$ if we bound the treelet size). The fact that this give a complexity worse than linear with respect to the database size might seem a concern, but this is actually only because we are retrieving more and more different types of treelets. The cost of retrieving one treelet remain linear with respect to the size of the corpus. We empirically find that even for very large values of N_C, processing time remain very reasonable (see next section).

It should be also noted that the constant hidden in the big-O notation can be (almost) arbitrarily reduced by precomputing more and more of the most common (and more expensive) treelets (a time-memory trade-off).

5 Experiments

We conducted experiments on a large database of 2.9 million automatically parsed dependency trees, with a total of nearly 60 million nodes[1]. The largest trees in the database have around 100 nodes. In order to see how performance scale with the size of the database, we also used a smaller subset of 230,000 trees containing near 6 million nodes.

We computed, using our algorithm, 100 randomly selected query trees having from 10 to 70 nodes, with an average of 27 nodes per tree. Table 5 shows the average performances per sentence. Considering the huge size of the database, a processing time below 1 second seems reasonable. The increase in processing time between the small and the large database is in line with the explanations of section 4.4. Path-to-Root Arrays are slightly better than Inverted indexes (we suspect a better implementation could increase the difference further). Both methods use up about the same disk space: around 500MB. We also find that the approach of section 4.2 brings virtually no overhead and gives similar performances whether the query tree is in the database or not (effectively reducing the worst-case computation time from days to seconds).

We also conducted a small English-to-Japanese translation experiment with a simple translation system using Synchronous Tree Substitution Grammars (STSG) for translating dependency trees. The system we used is still in an experimental state and probably not quite at the state-of-the-art level yet. However, we considered it was good enough for our purpose, since we mainly want to test our algorithm is a practical way. As a baseline, from our corpus of 2.9 millions dependency trees, we automatically extracted STSG rules of size smaller than 6 and stored them in a database, considering that extracting rules of larger sizes would lead to an unmanageable database size. We compared MT results using only the rules of size smaller than 6 to using all the rules computed on-the-fly after treelet retrieving by our method. These results are summarized on figure 6.

6 Flexible matching

We now describe an extension of the algorithm for approximate matching of treelets. We consider that each node of the query tree and database is labeled by 2 labels (or more) of different generality. For concreteness, let's consider dependency trees whose nodes are labeled by words and the Part-Of-Speech (POS) of these words. We want to retrieve treelets

[1]This database was an aggregate of several Japanese-English corpora, notably the Yomiuri newspaper corpus (Utiyama and Isahara, 2003) and the JST paper abstract corpus created at NICT(www.nict.go.jp) through (Utiyama and Isahara, 2007).

that match by word or POS with the query tree.

6.1 Processing multi-Label trees

To do this, the inverted index will just need to include entries for both words and POS. For example, the dependency tree "likes,V:1 (Paul,N:0 Salmon,N:2 (and,CC:3 (Tuna,N:4)))" would produce the following *(node,parents)* entries in the inverted index: {N:[(0,1) (2,1) (4,3)], Paul:[(0,1)], Salmon:[(2,1)],...}. This allows to search for a treelet containing any combination of labels, like "likes(N Salmon(CC(N)))".

We actually want to compute all of the treelets of a query tree $\mathcal{T}_Q$ labeled by words and POS (meaning each node can be matched by either word or POS).

We can compute $\mathcal{T}_Q$ without redundant computations by slightly modifying the algorithm 2. First, we modify the recursive representation of a treelet so that it also includes the chosen label of its root node. Then, the only modifications needed in algorithm 2 are the following: 1- at initialization (line 3), the elementary treelets corresponding to every possible labels are added to the discovered treelets set $\mathcal{D}$; 2- in procedure *compute-supertrees*, at line 8, we generate one ascending supertreelet per label.

6.2 Weighted search

While the previous method would allow us to compute as efficiently as possible all the treelets included in a multi-labeled query tree, there is still a problem: even avoiding redundant computations, the number of treelets to compute can be huge, since we are computing all combinations of labels. For each treelet of size m we would have had in a single label query tree, we now virtually have 2^m treelets. Therefore, it is not reasonable in general to try to compute all these treelets.

However, we are not really interested in computing all possible treelets. In our case, the POS labels allow us to retrieve larger examples when none containing only words would be available. But we still prefer to find examples matched by words rather than by POS. We therefore need to tell the algorithm that some treelets are more important that some others. While we have used the Computation Hypertree representation to compute treelets efficiently, we can also use it to prioritize the treelets we want to compute. This is easily implemented by giving a weight to every treelet. We can then modify our traversal strategy of the Inclusion DAG to compute treelets having the biggest weights first: we just need to specify that the treelet popped out on line 3 is the treelet with the highest score (more generally, we could consider a A* search).

POS matchings	Without	With
Processing time	0.9 s	22 s
Largest non-empty treelet size	8.7	11.4
Treelets of size>8	0.4	102
BLEU	12.0%	12.1%

Figure 7: Effect of POS-matching

6.3 Experiments

Using the above ideas, we have made some experiments for computing query dependency trees labeled with both words and POS. We score the treelets by giving them a penalty of -1 for each POS they contain, and stop the search when all remaining treelets have a score lower than -2 (in other words, treelets are allowed at most 2 POS-matchings). We also require POS-matched nodes to be non-adjacent.

We only have some small modifications to do to algorithm 2. In line 3 of algorithm 2, elementary treelets are assigned a weight of 0 or -1 depending on whether their label is a word or POS. Line 5 is replaced by "pop the first treelet with minimal weight and break the loop if the minimal weight is inferior to -2". In *compute-supertreelets*, we give a weight to the generated supertreelets by combining the weights of the child treelets.

Table 7 shows the increase in the size of the biggest non-empty treelets when allowing 2 nodes to be matched by POS. It also shows the impact on BLEU score of using these additional treelets for on-the-fly rule generation in our simple MT system. Improvement on BLEU is limited, but it might be due to a very experimental handling of approximately matched treelet examples in our MT system.

The computation time, while manageable, was much slower than in the one-label case. This is due to the increased number of treelets to be computed, and to the fact that POS-labeled elementary treelets have a high number of occurrences. It would be more efficient to use more specific labeling (e.g V-

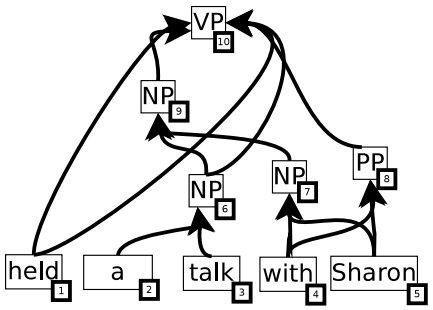


Figure 8: A packed forest.

mvt for verbs of movement instead of V).

7 Additional extensions

We briefly discuss here some additional extensions to our algorithm that we will not detail for lack of room and practical experiments.

7.1 Packed forest

Due to parsing ambiguities and automatic parsers errors, it is often useful to use multiple parses of a given sentence. These parses can be represented by a packed forest such as the one in figure 8. Our method allows the use of packed representation of both the query tree and the database.

For the inverted index, the only difference is that now, an occurrence of a label can have more than one parent. For example, the inverted index of a database containing the packed forest of figure 8 would contain the following entries: {held: [(1,10a),(1,10b)], NP: [(6,9),(7,9),(9,10a)], VP:[(10,N)], PP:[(8,10b)], a:[(2,6)], talk:[(3,6)], with:[(4,7) (4,8)], Sharon:[(5,7) (5,8)]}. Where 10a and 10b are some kind of virtual position that help to specify that *held* and NP_8 belong to the same children list. We could also include a cost on edges in the inverted index, which would allow to prune matchings to unlikely parses.

The inverted index can now be used to search in the trees contained in a packed forest database without any modification. Modifications to the algorithm in order to handle a packed forest query are similar to the ones developed in section 6.

7.2 Discontinuous treelets

As we discussed in section 2.3, using a representation for the position of every node similar to (Zhang et al., 2001), it is possible to determine the distance and ancestor relationship of two nodes by just comparing their positions. This opens the possibility of computing the occurrences of discontinuous treelets in much the same way as is done in (Lopez, 2007) for discontinuous substrings. We have not studied this aspect in depth yet, especially since we are not aware of any MT system making use of discontinuous syntax tree examples. This is nevertheless an interesting future possibility.

8 Related work

As we previously mentioned, (Lopez, 2007) and (Callison-Burch et al., 2005) propose a method similar to ours for the string case.

We are not aware of previous proposals for efficient on-the-fly retrieving of translation examples in the case of Syntax-Based Machine Translation. Among the works involving rule precomputation, (Zhang et al., 2009) describes a method for efficiently matching precomputed treelets rules. These rules are organized in a kind of prefix tree that allows efficient matching of packed forests. (Liu et al., 2006) also propose a greedy algorithm for matching TSC rules to a query tree.

9 Conclusion and future work

We have presented a method for efficiently retrieving examples of treelets contained in a query tree, thus allowing on-the-fly computation of translation rules for Syntax-Based systems. We did this by building on approaches previously proposed for the case of string examples, proposing an adaptation of the concept of Suffix Arrays to trees, and formalizing computation as the traversal of an hypergraph. This hypergraph allows us to easily formalize different computation strategy, and adapt the methods to flexible matchings. We still have a lot to do with respect to improving our implementation, exploring the different possibilities offered by this framework and proceeding to more experiments.

Acknowledgments

We thank the anonymous reviewers for their useful comments.

References

R. Baeza-Yates and A. Salinger. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In *String Processing and Information Retrieval*, page 1324.

N. Bruno, N. Koudas, and D. Srivastava. 2002. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, page 310321.

C. Callison-Burch, C. Bannard, and J. Schroeder. 2005. Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 255–262. Association for Computational Linguistics Morristown, NJ, USA.

David Chiang. 2007. Hierarchical Phrase-Based translation. *Computational Linguistics*, 33(2):201–228, June.

M. Dubiner, Z. Galil, and E. Magen. 1994. Faster tree pattern matching. *Journal of the ACM (JACM)*, 41(2):205213.

P. Koehn, F. J. Och, and D. Marcu. 2003. Statistical phrase-based translation. In *Proceedings of HLT-NAACL*, pages 48–54. Association for Computational Linguistics.

Z. Liu, H. Wang, and H. Wu. 2006. Example-based machine translation based on tree–string correspondence and statistical generation. *Machine translation*, 20(1):25–41.

A. Lopez. 2007. Hierarchical phrase-based translation with suffix arrays. In *Proc. of EMNLP-CoNLL*, page 976985.

U. Manber and G. Myers. 1990. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, San Francisco, CA, USA. Society for Industrial and Applied Mathematics Philadelphia, PA, USA.

H. Mi, L. Huang, and Q. Liu. 2008. Forest based translation. *Proceedings of ACL-08: HLT*, page 192199.

Toshiaki Nakazawa and Sadao Kurohashi. 2008. Syntactical EBMT system for NTCIR-7 patent translation task. In *Proceedings of NTCIR-7 Workshop Meeting*, Tokyo, Japon.

C. Quirk, A. Menezes, and C. Cherry. 2005. Dependency treelet translation: Syntactically informed phrasal SMT. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, page 279.

M. Utiyama and H. Isahara. 2003. Reliable measures for aligning japanese-english news articles and sentences. In *Proceedings of ACL*, pages 72–79, Sapporo,Japon.

M. Utiyama and H. Isahara. 2007. A japanese-english patent parallel corpus. In *MT summit XI*, pages 475–482.

M. Yamamoto and K. W. Church. 2001. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30.

Y. Zhang and S. Vogel. 2005. An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora. In *Proceedings of EAMT*, pages 294–301, Budapest, Hungary.

C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. 2001. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, page 425436.

H. Zhang, M. Zhang, H. Li, and Chew Lim Tan. 2009. Fast translation rule matching for syntax-based statistical machine translation. In *Proc. of EMNLP*, pages 1037–1045.