Submitted 06/2014; Published 12/2014

Extraction of Phrase-Structure Fragments with a Linear Average Time Tree-Kernel

Andreas van Cranenburgh[*][†] ANDREAS.VAN.CRANENBURGH@HUYGENS.KNAW.NL

[*]*Huygens Institute for the History of the Netherlands, Royal Netherlands Academy of Arts and Sciences*
[†]*Institute for Logic, Language and Computation, University of Amsterdam*

Abstract

We present an algorithm and implementation for extracting recurring fragments from treebanks. Using a tree-kernel method the largest common fragments are extracted from each pair of trees. The algorithm presented achieves a thirty-fold speedup over the previously available method on the Wall Street Journal dataset. It is also more general, in that it supports trees with discontinuous constituents. The resulting fragments can be used as a tree-substitution grammar or in classification problems such as authorship attribution and other stylometry tasks.

1. Introduction

Treebanks are a rich source of lexical and structural patterns. A simple and common approach is to consider the frequencies of individual grammar productions; the main example being treebank grammars for parsing (Charniak, 1996), but also stylometry (cf. Baayen et al., 1996; Raghavan et al., 2010; Ashok et al., 2013). Richer patterns involve multiple lexical items or constituents; i.e., they may consist of the co-occurrence of a sequence of productions that make up a specific phrase or a grammatical construction. Kernel methods, which quantify similarity by decomposing a signal into components, and specifically tree kernel methods (Collins and Duffy, 2001, 2002), consider such patterns but obtain only a numeric value about the degree of similarity of structures,[1] without making explicit what the structures have in common. The usefulness of extracting explicit fragments, however, is underscored by one of the conclusions of Moschitti et al. (2008, p. 222):

> The use of fast tree kernels (Moschitti, 2006a) along with the proposed tree representations makes the learning and classification much faster, so that the overall running time is comparable with polynomial kernels. However, when used with [Support Vector Machines] their running time on very large data sets (e.g., millions of instances) becomes prohibitive. Exploiting tree kernel-derived features in a more efficient way (e.g., by selecting the most relevant fragments and using them in an explicit space) is thus an interesting line of future research.

Aside from their use as features in machine learning tasks, tree fragments also have applications in computational linguistics for statistical parsing and corpus linguistics.

Since we will focus on the problem of finding the largest common fragments in tree pairs, there is an intuitive relation to the problem of finding all longest common subsequences of a string pair. However, in the case of tree structures the problem is more constrained than with sequences, since any matching nodes must be connected through phrase-structure.

An algorithm for extracting recurring phrase-structure fragments was first presented by Sangati et al. (2010). Their algorithm is based on a Quadratic Tree Kernel that compares each node in the input to all others, giving a quadratic time complexity with respect to the number of nodes in the treebank. Moschitti (2006b) presents the Fast Tree Kernel, which operates in linear average time. However, his algorithm only returns a list

1. In fact, the practice of using a kernel to quantify similarity without making it explicit is referred to as the 'kernel trick' in the machine learning literature.

of matching nodes. This work presents an algorithm that exploits the Fast Tree Kernel of Moschitti (2006b) to extract recurring fragments, providing a significant speedup over the quadratic approach. Our implementation is available for download, including source code, cf. `https://github.com/andreasvc/disco-dop`

2. Applications, related work

The two main applications of tree fragment extraction so far are in parsing and classification problems.

Tree fragments can be used as grammar productions in Tree-Substitution Grammars (TSG). TSGs are used in the Data-Oriented Parsing framework; DOP (Scha, 1990; Bod, 1992). In Data-Oriented Parsing the treebank is considered as the grammar, from which all possible fragments can in principle be used to derive new sentences through tree-substitution. Grammar induction is therefore conceptually straightforward (although the grammar is very large), as there is no training or learning involved. This maximizes re-use of previous experience.

Since representing all possible fragments of a treebank is not feasible (their number is exponential in the number of nodes), one can resort to using a subset, but sampling or arbitrary restrictions are likely to lead to a suboptimal set of fragments, since the vast majority of fragments occur only once in the treebank (Sangati et al., 2010). Double-DOP; 2DOP (Sangati and Zuidema, 2011) avoids this by restricting the set to fragments that occur at least twice. The heuristic of this model is to construct the grammar by extracting the largest common fragments for every pair of trees, just as the tool presented in this work. A recent implementation generalizes this model to trees with discontinuous constituents (van Cranenburgh and Bod, 2013).

An alternative to the all-fragments assumption of DOP takes a Bayesian approach to selecting fragments and assigning probabilities. Bayesian TSGs (O'Donnell et al., 2009; Post and Gildea, 2009; Cohn et al., 2010; Shindo et al., 2012) are induced by sampling fragments using Markov Chain Monte Carlo (MCMC) following a Zipfian long tail distribution.

Aside from the generative use of fragments in TSGs, tree fragments are also used for discriminative re-ranking. Implicit fragment features (counted but not extracted) are used in Collins and Duffy (2001, 2002) through a tree kernel, and explicit fragment features are used in Charniak and Johnson (2005, p. 178: HeadTree and NGramTree features). Another discriminative application of recurring fragments is in text classification tasks. Common tree fragments can be used to define a similarity measure between texts, which can be applied to the task of authorship attribution (van Cranenburgh, 2012c). In the latter case the efficiency of extracting fragments has been exploited by extracting fragments at classification time, i.e., a memory-based approach, without defining features in advance. Other classification tasks have been modeled with fragments induced by Bayesian TSGs: e.g., native language detection (Swanson and Charniak, 2012), stylometry of scientific papers (Bergsma et al., 2012), and corpus analysis of source code (Allamanis and Sutton, 2014).

Finally, there is a tradition in the data mining literature called frequent tree mining or frequent structure mining (Jiménez et al., 2010). This tradition does not apply the maximal fragment constraint and explores different kinds of fragments and trees. The approach is based on the Apriori algorithm and works by generating candidates that occur with a specified minimum frequency. However, the relaxation of the maximality constraint results in an exponential number of fragments, while for linguistic applications the focus on frequent fragments is arguably in conflict with the importance of the long tail in language, specifically, the importance of low frequency events. See Martens (2010) though, which applies a frequent tree mining approach to unordered dependency trees.

3. Definitions

The notion of a tree fragment is defined as follows:

Definition 1 A *fragment* f of a tree T is a connected subgraph of T, such that f contains at least 2 nodes, and each node in f either is an empty node (a substitution site), or immediately dominates the same immediate children as the corresponding node in T.

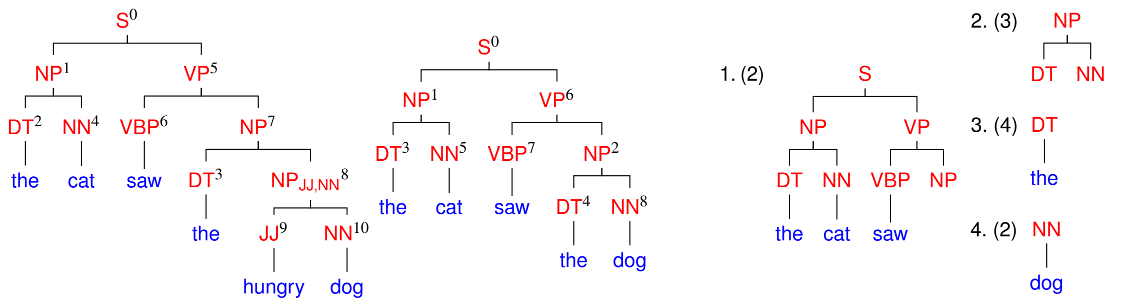


Figure 1: Left: two example trees containing recurring patterns; the superscript denotes an identifier for each node. Right: all maximal fragments in the pair of trees on the left (frequencies in parentheses). Note that the second fragment is a subgraph of the first fragment, but is still a maximal fragment when considering $\langle NP^1, NP^2\rangle$ of the first and second tree, respectively.

In this paper we identify a non-terminal node of a tree with its (grammar) production. The production at a non-terminal node is a tuple of labels of that node and its immediate children. A label is either the phrase label of a non-terminal node, or the lexical item of a terminal node.

Note that in Moschitti (2006a), this fragment definition corresponds to the subset tree (SST) kernel. The SST kernel can be contrasted with the subtree (ST) kernel where all descendants of a subtree are retained, and the partial tree (PT) kernel where nodes may include a subsequence of the children of the corresponding node in the original tree. In this work we only address the SST kernel.

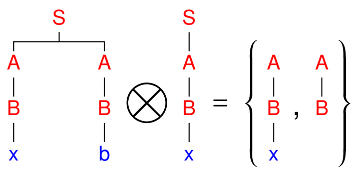
The subset of possible recurring fragments we consider is defined by the largest overlapping fragments for any pair of trees in the input:

Definition 2 Given a pair of trees $\langle a, b\rangle$, and a pair of nodes $\langle p, q\rangle$ such that p is a node in a, q is a node in b, with p and q having the same production, the *maximal common fragment* defined by $\langle p, q\rangle$ is the largest fragment f that occurs in a incorporating node p and occurs in b incorporating node q.

Note that our definition of maximal common fragment differs from the one in Sangati et al. (2010, sec. 2):

> "A (partial) fragment τ shared between [two] trees is maximal if there is no other shared (partial) fragment starting with the same node as in τ and including all its nodes."

Furthermore, in their formulation, maximality of fragments is determined with respect to the first tree in the comparison (cf. Sangati et al., 2010, Algorithm 2, second statement). Our formulation determines maximality with respect to node pairs, i.e., involving both trees. A consequence of this is that for the algorithm of Sangati et al. (2010), the order of the trees in the input can have an effect on the result; i.e., the extraction of fragments from a node pair is not a commutative operation. Consider a pair of trees and the fragments that `FragmentSeeker` (Sangati et al., 2010) extracts from them:



When the order of the input is reversed, the second fragment is not extracted, because it is a subset of the first. With our algorithm following definition 2, the two fragments are extracted regardless of the order of the input.

We are also interested in the frequencies of fragments. The frequency is not defined with respect to occurrences as maximal common fragment, but with respect to all occurrences:

Definition 3 The *occurrence count* of a fragment in a treebank is the total number of occurrences of a fragment in a collection of trees.

As an example of the preceding definitions, see figure 1, which shows two trees and their maximal common fragments with occurrence counts.

4. Fragment extraction with tree kernels

In this section we first discuss the Fast Tree Kernel, followed by the algorithm to extract fragments from the matching nodes it finds. We then discuss two extensions that find occurrence counts and handle discontinuous constituents.

Algorithm 1 lists the pseudocode for the Fast Tree Kernel (FTK) by Moschitti (2006b). The pseudocode for extracting fragments is shown in algorithm 2; the main entry point is FUNCTION recurring-fragments. This formulation is restricted to binary trees, without loss of generality, since trees can be binarized into a normal form and debinarized without loss of information.

Sangati et al. (2010) define the extraction of fragments as considering all pairs of trees from a single treebank. Since it may also be interesting to investigate the commonalities of two different treebanks, we generalize the task of finding recurring fragments in a treebank to the task of finding the common fragments of two, possibly equal, treebanks. In case the treebanks are equal, only half of the possible tree pairs have to be considered: the fragments extracted from $\langle t_n, t_m\rangle$, with $n < m$, are equal to those of $\langle t_m, t_n\rangle$.

```
INPUT: A pair of trees $\langle a, b\rangle$, with the nodes of each tree sorted by a common ordering on their productions.
    $a[i]$ returns the $i$-th production in $a$.
OUTPUT: A boolean matrix $\mathcal{M}$ with $\mathcal{M}[i, j]$ true iff the production at $a[i]$ equals the one at $b[j]$.
 1: FUNCTION fast-tree-kernel($a, b$)
 2:     $i \leftarrow 0, j \leftarrow 0$
 3:     $\mathcal{M} \leftarrow |a| \times |b|$ boolean matrix, values initialized as false.
 4:     WHILE $i < |a| \wedge j < |b|$
 5:         IF $a[i] < b[j]$
 6:             $j \leftarrow j + 1$
 7:         ELSE IF $a[i] > b[j]$
 8:             $i \leftarrow i + 1$
 9:         ELSE
10:             WHILE $a[i] = b[j]$
11:                 $j\prime \leftarrow j$
12:                 WHILE $a[i] = b[j\prime]$
13:                     $\mathcal{M}[i, j\prime] \leftarrow true$
14:                     $j\prime \leftarrow j\prime + 1$
15:                 $i \leftarrow i + 1$
```

Algorithm 1: The Fast Tree Kernel, adapted from Moschitti (2006b).

4.1 The Fast Tree Kernel

See algorithm 1 for the pseudocode of the Fast Tree Kernel. The insight that makes this kernel fast on average is that the problem of finding common productions can be viewed as the problem of finding the intersection of two multisets that have been sorted in advance. The input of the function fast-tree-kernel is a pair of trees

$\langle a, b \rangle$, that are represented as a list of nodes sorted by their productions.[2] The requirement for sorted input depends on an ordering defined over the grammar productions, but note that the nature of this ordering is irrelevant, as long as it is consistently applied (e.g., productions may be sorted lexicographically by the labels of the left and right hand sides of productions; or productions can be assigned a sequence number when they are first encountered). The output is a boolean matrix where the bit at (n, m) is set *iff* the nodes at those indices in the respective trees have the same production. From this table the bitvectors corresponding to fragments are collected and stored in the results table.

Given the trees from the introduction as input, the resulting set of matching node pairs can be seen as a matrix; cf. table 1. The matrix visualizes what makes the algorithm efficient: there is a diagonal path along which comparisons have to be made, but most node pairs (i.e., the ones with a different label) do not have to be considered. The larger the number of production types, the higher the efficiency. In case there is only a single non-terminal label, and hence only one phrasal, binary production, the efficiency of the algorithm degenerates and the worst-case quadratic complexity obtains.

	S^0	NP^1	NP^2	DT^3	DT^4	NN^5	VP^6	VBP^7	NN^8
S^0	1								
NP^1		1	1,2						
DT^2				1,3	3				
DT^3				3	3				
NN^4						4			
VP^5							1		
VBP^6								1	
NP^7									
$\mathrm{NP}^8_{\mathrm{JJ,NN}}$									
JJ^9									
NN^{10}									1

Table 1: Matrix when trees in figure 1 are compared. Highlighted cells are nodes with a common production. The numbers indicate the fragments extracted, corresponding with those in figure 1. Note that for expository purposes, the nodes are presented in depth-first order.

4.2 Extracting maximal connected subsets

After the matrix with matching nodes has been filled, we identify the maximal fragments that matching nodes are part of. We traverse the second tree in depth-first order, in search for possible root nodes of fragments; cf. algorithm 2, line 6.

The fast tree kernel in algorithm 1 returns matching node pairs as output. The resulting adjacency matrix $\mathcal{M}$ is iterated over in line 8. This may appear to be quadratic in the number of nodes of the trees, but only the cells for matching node pairs need to be visited, and each pair is visited once. Since it is possible to scan for cells with 1-bits efficiently (cf. sec. 5), the linear average time complexity is maintained.

Whenever a 1-bit corresponding to a matching node pair is encountered, a fragment with that node pair is extracted. Both trees are traversed in parallel, top-down from that node pair onwards, to collect the subset of nodes for the fragment; cf. algorithm 2, line 9. Both trees need to be considered to ensure that extracted

2. The requirement for sorted input does not affect the asymptotic complexity of the algorithm because each tree is sorted separately, so is not affected by the total number of nodes in the treebank. Furthermore, the sorting is done offline and only once.

```
INPUT: A treebank $TB$, with the nodes of each tree sorted by their productions.
OUTPUT: A set F with maximal recurring fragments from $TB$.
 1: FUNCTION recurring-fragments($TB$)
 2:     $F \leftarrow \emptyset$
 3:     FOR ALL $\langle a, b\rangle \in TB \times TB$ with $a \neq b$
 4:         $\mathcal{M} \leftarrow$ fast-tree-kernel($a, b$)
 5:         $F \leftarrow F \cup$ fragments($a, b$, root($b$), $\mathcal{M}$)

INPUT: $a, b$ are trees with $a[i]_l, a[i]_r$ the index of
    the left and right child of node $i$ of $a$ (mutatis mutandis for $b[j]_l$ and $b[j]_r$).
    $\mathcal{M}$ is a boolean matrix with $\mathcal{M}[i, j]$ true iff the production at $a[i]$ equals the one at $b[j]$.
OUTPUT: The set F of maximal fragments in $a$ and $b$.
 6: FUNCTION fragments($a, b, j, \mathcal{M}$) {Traverse tree $b$ top-down starting from $j$.}
 7:     $F \leftarrow \emptyset$
 8:     FOR ALL $i$ such that $\mathcal{M}[i, j]$
 9:         $F \leftarrow F \cup$ extract-at($a, b, i, j, \mathcal{M}$)
10:     IF has-left-child($b[j]$)
11:         $F \leftarrow F \cup$ fragments($a, b, b[j]_l, \mathcal{M}$)
12:     IF has-right-child($b[j]$)
13:         $F \leftarrow F \cup$ fragments($a, b, b[j]_r, \mathcal{M}$)

INPUT: Indices $i, j$ denoting start of a fragment in trees $a, b$.
OUTPUT: The fragment $f$ rooted at $a[i]$ and $b[j]$.
14: FUNCTION extract-at($a, b, i, j, \mathcal{M}$) {Traverse $a$ and $b$ top-down, in parallel.}
15:     $f \leftarrow$ tree-from-production($a[i]$) {Create a depth 1 subtree from a grammar production.}
16:     $\mathcal{M}[i, j] \leftarrow$ false {do not extract a fragment with this node pair again.}
17:     IF has-left-child($a[i]$) $\wedge$ $\mathcal{M}[a[i]_l, b[j]_l]$
18:         $f_l \leftarrow$ extract-at($f, a, b, a[i]_l, b[j]_l, \mathcal{M}$) {add as left subtree}
19:     IF has-right-child($a[i]$) $\wedge$ $\mathcal{M}[a[i]_r, b[j]_r]$
20:         $f_r \leftarrow$ extract-at($f, a, b, a[i]_r, b[j]_r, \mathcal{M}$) {add as right subtree}
```

Algorithm 2: Extract maximal recurring fragments given common nodes for each pair of trees in a treebank.

subsets are connected in both trees. Every node pair that is visited contributes a subtree corresponding to the production at the node pair, which is added to the fragment that is being constructed. The node pair is marked such that it will not be used in another fragment.

Note that the algorithm of Sangati et al. (2010) combines the tree kernel and extraction of maximal connected subsets in a single pass, and is thus a dynamic programming approach. That is, as the algorithm iterates over possibly matching pairs of nodes, a new fragment may result which subsumes a fragment extracted at an earlier stage. Our approach is able to use a greedy algorithm for finding maximal subgraphs by using a two pass approach consisting of first finding all matching nodes, and then extracting fragments.

4.3 Occurrence counts

It is possible to keep track of the number of times a fragment is extracted. Because of the maximality constraint, this count will typically be a lower bound on the true occurrence count of a fragment. The true occurrence count may be useful for a probabilistic model or for corpus analysis. To do this, a second pass needs to be made over the treebank, in which for each fragment, all its occurrences are counted—including occurrences that are not maximal for any tree pair.

Counting exploits an index listing the set of trees which contain a given production. By taking the intersection of the sets for all productions in a fragment, a set of candidate trees can be composed efficiently.

These candidates will contain all of the productions in the fragment, but it is necessary to traverse the candidates exhaustively to confirm that the productions are present in the right configuration corresponding to the fragment.

Aside from obtaining counts, it is also possible to obtain for each fragment a vector of indices of all trees that contain the fragment. This makes it possible to address diachronic questions, if the trees in the input are presented in chronological order.

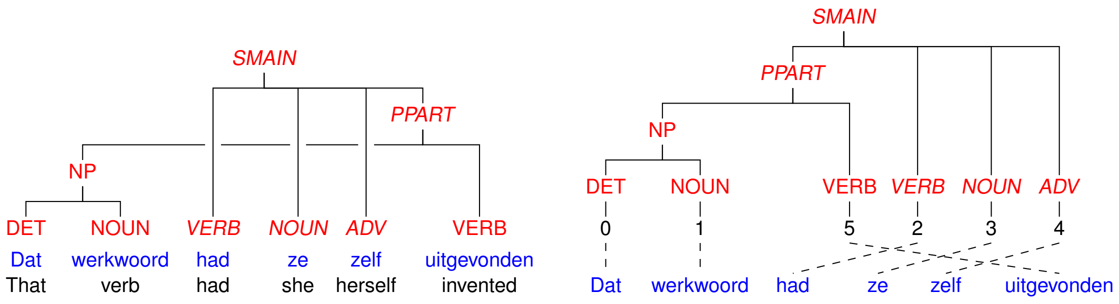


Figure 2: An illustration of the representation for trees with discontinuous constituents. Shown on the left is the original discontinuous tree from the Alpino treebank (van der Beek et al., 2002). On the right is a version in which the phrase structure has been decoupled from its surface form using indices. Translation: *That verb she had invented herself.*

4.4 Trees with discontinuous constituents

Our implementation supports trees with discontinuous constituents. A discontinuous constituent is a constituent whose yield does not consist of a single contiguous string, but rather a tuple of strings. As a simple example in English, consider:

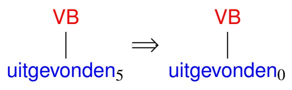
(1) Wake your friend up

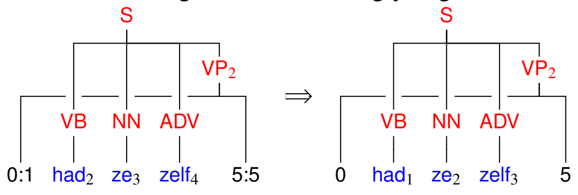
(2) [$_{\text{VP}}$ Wake . . . up]

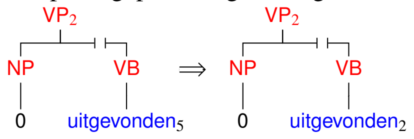
Where the phrasal verb in (1) may be said to constitute the discontinuous constituent (2).

The first treebank that introduced discontinuous constituents as a major component of its annotation is the German Negra treebank (Skut et al., 1997). Syntax trees in this style of annotation are defined as unordered trees, with the caveat that there is a total ordering of the words in the original sentence. This caveat is important because for example in the case of the problem of calculating the tree-edit distance between trees, the problem is tractable for ordered trees, but not for unordered trees (Zhang et al., 1992).

We use a transform-backtransform approach which makes it possible to use the fragment extraction algorithm without further modification. In order to use the unordered trees from treebanks with discontinuous constituents, we use the ordering of the words in the sentences to induce a canonical order for the internal nodes of the tree. This makes it possible to use the same data structures as for continuous trees. We use a representation where leaf nodes are decorated with indices indicating their position in the sentence (cf. figure 2). Using the indices a canonical order for internal nodes is induced based on the lowest index dominated by each node. Indices are used not only to keep track of the order of lexical nodes, but also to store where contributions of frontier non-terminals end up relative to the contributions of other non-terminals. This is necessary in order to preserve the configuration of the yield in the original sentence. The indices are based on those in the original sentence, but need to be decoupled from this original context, so that the indices of a fragment are independent from the position in which the fragment occurred in the original tree. This process of canonicalization is analogous to how a production of a Linear Context-Free Rewriting System (LCFRS) can be read off from a tree with discontinuous constituents (Maier and Søgaard, 2008), where intervals of indices are replaced by variables. The canonicalization of fragments is achieved in three steps:

1. Translate indices so that they start at 0; e.g.:


2. Reduce spans of frontier non-terminals to length 1; move surrounding indices accordingly; e.g.:


3. Compress gaps to length 1; e.g.:


As an example of this procedure, consider two variants of a famous quotation attributed to Churchill, mocking the notion that sentences should not end with a preposition:

(3) a. This is the sort of English up with which I will not put.
b. Ending a sentence with a preposition is the sort of English up with which I will not put.

If we apply an analysis with discontinuous constituents, these sentences contain a common verb phrase as an argument to the verb phrase "I will not", namely (4-a) and (4-b). If these are canonicalized, they map to the same fragment (4-c):

(4) a. [$_{\text{VP}}$ up$_6$ with$_7$ which$_8$ . . . put$_{12}$]
b. [$_{\text{VP}}$ up$_{11}$ with$_{12}$ which$_{13}$ . . . put$_{17}$]
c. [$_{\text{VP}}$ up$_0$ with$_1$ which$_2$ . . . put$_4$]

5. Implementation

A number of strategies have been employed to implement the fragment extraction algorithm as efficiently as possible. We use the Cython programming language (Behnel et al., 2011) which is a superset of the Python language that translates to C code and compiles to Python extension modules that seamlessly integrate with larger Python applications. We use manual memory management for the treebank and temporary arrays of the algorithm. This avoids the overhead of garbage collection and the requirement that all values must reside in an object ('boxed') as in Python and other managed languages.

A tree is represented as an array of tightly packed `structs` for each node, with the grammar production represented as an integer ID, and child nodes using array indices. Mapping productions to integer IDs ensures that comparisons between productions are cheap. In contrast to a pointer-based tree, traversing this array representation does not require indirection and has good memory locality.

When a pair of trees is compared, the results are stored in a bit matrix. The operations that need to iterate over set bits exploit CPU instructions to do this efficiently on one machine word (typically 64 bits) at a time (e.g., the 'find first set' instruction that finds the index of the first set bit of an integer). Fragments are first stored as a bitvector of a given tree, where each bit refers to the presence or absence of a particular production in the tree. These bitvectors are later converted to a string with the fragment in bracket notation. This ensures that fragments as extracted from different trees are recognized as equivalent and it is the format in which the results are returned.

The problem of extracting fragments from a treebank is a so-called *embarrassingly parallel* problem. This means that the work on a single tree pair is not affected by any other part of the treebank, and computations can be trivially distributed over any available computing power. In order to exploit multiple processing cores, we use the Python `multiprocessing` library. After reading the treebank the tree pairs are distributed evenly over a pool of processes, after which the results are collected.

		Number of		Time (hr:min)	
Method	Corpus	Trees	Fragments	Wall	CPU
Sangati et al. (2010):					
QTK	WSJ 2–21	39,832	990,156	8:23	124:04
This work:					
FTK	WSJ 2–21	39,832	990,890	0:18	4:01
FTK	Negra, train set	18,602	176,060	0:04	0:32
FTK	Gigaword, NYT 1999–11	502,424	9.7 million	9:54	~ 160

Table 2: Performance comparison of fragment extraction with the Quadratic Tree Kernel (QTK) and the Fast Tree Kernel (FTK) based algorithm. Wall clock time is when using 16 cores.

6. Benchmark

As a benchmark we use the training sections of the WSJ and Negra treebanks (Marcus et al., 1993; Skut et al., 1997), and a section of the Annotated Gigaword (Napoles et al., 2012, `nyt_eng_199911`), to validate that the more efficient algorithm makes it possible to handle larger treebanks. The WSJ treebank is binarized with $h = 1$, $v = 2$ markovization (Klein and Manning, 2003) and stripped of traces and function tags. To demonstrate the capability of working with discontinuous treebanks, we use the German Negra treebank, also binarized with $h = 1$, $v = 2$ markovization, and punctuation re-attached as described in van Cranenburgh (2012a). The Gigaword section is binarized without markovization. Work is divided over 16 CPU cores to demonstrate that the algorithm lends itself to parallelization.

See table 2 for a performance comparison. For the Sangati et al. (2010) implementation we use the implementation published as part of Sangati and Zuidema (2011). Because we use the markovization setting of $h = 1$, $v = 2$ described above, we get a larger number of fragments and longer running time than the results for the non-binarized WSJ treebank in Sangati et al. (2010). The times reported include the work for obtaining occurrence counts of fragments.[3] Note that there is a slight difference in the number of fragments found, but both implementations agree on 99.93 % of fragments found in the WSJ treebank. This difference is due to the difference in definition of what constitutes a maximal fragment discussed in section 3. The figure of 124 hours for the WSJ treebank with the implementation of Sangati et al. (2010) might seem large, as the training set of WSJ is not particularly large, at about 40K sentences, and quadratic algorithms are typically considered reasonable. However, note that the algorithm is quadratic with respect to the number of nodes in the treebank. After binarization, there are 2,045,118 nodes in the training set of WSJ, and the square of that number is considerable.

The Fast Tree Kernel delivers a substantial asymptotic improvement: we obtain a 30-fold speedup over Sangati et al. (2010). This speedup opens up the possibility of applying fragment extraction to much larger

3. Erratum: an earlier report on this work (van Cranenburgh, 2012b) reported run times without occurrence counts, while the results of Sangati et al. (2010) include them, which exaggerated the speedup.

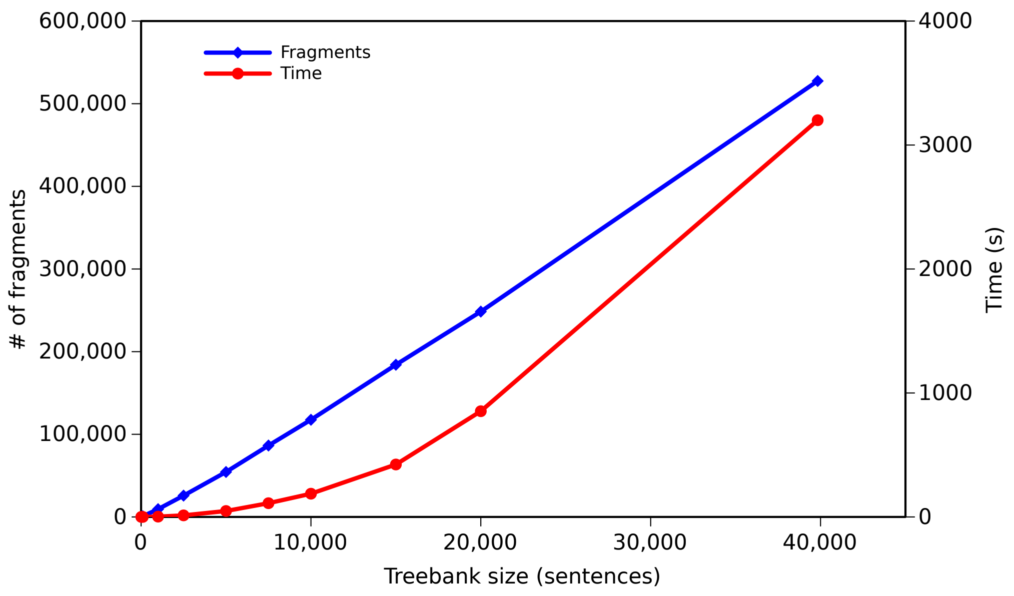


Figure 3: A plot of the running time and the number of fragments as a function of the number of trees in the input. The input is the Penn treebank, Wall Street Journal section 2–21, binarized $h = \infty, v = 1$; without extracting occurrence counts.

corpora. In addition to the improvement in runtime, the memory usage is eight times lower (less than 1 GB per process).

Part of this speedup is attributable to the more low-level style of programming using bitvectors and tightly packed data structures. However, in earlier experiments with a re-implementation of the algorithms specified by Sangati et al. (2010) with these optimizations, only a two-fold speedup was achieved. Therefore, most of the speedup comes from the Fast Tree Kernel and the greedy depth-first fragment extraction introduced in this work.

About 8.5 % of fragments extracted from the Negra treebank contain a discontinuous root or internal node, compared to 30 % of sentences in the treebank that contain one or more discontinuous constituents.

When the fragment extraction is applied to a larger number of trees from Gigaword, the extraction of fragments is still feasible, as long as occurrence counts are not requested, which does not scale as well as the other parts in terms of run time. This is because obtaining the counts requires iterating over all fragments and trees, both of which grow proportionately with treebank size. Future work should focus on optimizing this aspect of the algorithm by improving the indexing of the treebank. Perhaps this can be done using a variant of a suffix array. A suffix array of a corpus allows for arbitrary substring search in time linear to the length of the substring (Abouelhoda et al., 2004). Suffix arrays have been used for machine translation (Lopez, 2007). However, suffix arrays only deal with strings. They would either need to be generalized to tree structures, or adapted as a filter to speed up finding trees with matching terminals. Alternatively, it may be the case that for the application of Probabilistic Tree-Substitution Grammars, approximate frequencies suffice.

Figure 3 plots the number of fragments and run time as the number of trees is increased. It is clear that the number of fragments extracted grows linearly with the number of trees. The plot of the running time appears close to linear, with a slight curvature (similar to the graphs in Moschitti 2006a). While the tree kernel is linear average time in the number of nodes, the number of tree pairs that are compared is quadratic in the number of trees, when all tree pairs are considered. The complexity can be limited further by only considering tree pairs with a certain number of overlapping lexical items, or looking only at a sample of tree pairs. A further optimization, suggested in Collins and Duffy (2001, sec. 5) and implemented in Aiolli et al. (2007), is to exploit common subtrees in the input by representing the set of trees as a single directed acyclic graph. Rieck et al. (2010) present an Approximate Tree Kernel which attains a large speedup by significantly reducing the space of fragments that are considered.

7. Conclusion

We have presented a method and implementation for fragment extraction using an average case linear time tree kernel. We obtain a substantial speedup over the previously presented quadratic-time algorithm, and the resulting fragments and frequencies have been validated against the output of the latter. Additionally, we introduced support for discontinuous constituents.

The fragment extractor including source code is available for download as part of `disco-dop`, cf. `https://github.com/andreasvc/disco-dop`

Acknowledgements

I am grateful to Federico Sangati for making the source code of his `FragmentSeeker` available, to Remko Scha for extensive comments, to Rens Bod for commenting on a previous version of this article, and to the anonymous reviewers for useful feedback. This work is part of The Riddle of Literary Quality, a project supported by the Royal Netherlands Academy of Arts and Sciences through the Computational Humanities Program.

References

Abouelhoda, Mohamed Ibrahim, Stefan Kurtz, and Enno Ohlebusch (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86.

Aiolli, Fabio, Giovanni Da San Martino, Alessandro Sperduti, and Alessandro Moschitti (2007). Efficient kernel-based learning for trees. In *Proceeding of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 308–315.

Allamanis, Miltiadis and Charles Sutton (2014). Mining Idioms from Source Code. ArXiv e-print. `http://arxiv.org/abs/1404.0417`.

Ashok, Vikas, Song Feng, and Yejin Choi (2013). Success with style: Using writing style to predict the success of novels. In *Proceedings of EMNLP*, pages 1753–1764. `http://aclweb.org/anthology/D13-1181`.

Baayen, Harold, H. van Halteren, and F. Tweedie (1996). Outside the cave of shadows: Using syntactic annotation to enhance authorship attribution. *Literary and Linguistic Computing*, pages 121–132.

Behnel, Stefan, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith (2011). Cython: The best of both worlds. *Computing in Science and Engineering*, 13:31–39.

Bergsma, Shane, Matt Post, and David Yarowsky (2012). Stylometric analysis of scientific articles. In *Proceedings of NAACL*, pages 327–337. `http://aclweb.org/anthology/N12-1033`.

Bod, Rens (1992). A computational model of language performance: Data-oriented parsing. In *Proceedings COLING*, pages 855–859. `http://aclweb.org/anthology/C92-3126`.

Charniak, Eugene (1996). Tree-bank grammars. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1031–1036.

Charniak, Eugene and Mark Johnson (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of ACL*, pages 173–180. `http://aclweb.org/anthology/P05-1022`.

Cohn, Trevor, Phil Blunsom, and Sharon Goldwater (2010). Inducing tree-substitution grammars. *The Journal of Machine Learning Research*, 11(Nov):3053–3096.

Collins, Michael and Nigel Duffy (2001). Convolution kernels for natural language. In *Advances in neural information processing systems*, pages 625–632.

Collins, Michael and Nigel Duffy (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of ACL*. `http://aclweb.org/anthology/P02-1034`.

Jiménez, Aí da, Fernando Berzal, and Juan-Carlos Cubero (2010). Frequent tree pattern mining: A survey. *Intelligent Data Analysis*, 14(6):603–622.

Klein, Dan and Christopher D. Manning (2003). Accurate unlexicalized parsing. In *Proceedings of ACL*, volume 1, pages 423–430. `http://aclweb.org/anthology/P03-1054`.

Lopez, Adam (2007). Hierarchical phrase-based translation with suffix arrays. In *Proceedings of EMNLP-CoNLL*, pages 976–985. `http://aclweb.org/anthology/D07-1104`.

Maier, Wolfgang and Anders Søgaard (2008). Treebanks and mild context-sensitivity. In *Proceedings of Formal Grammar 2008*, pages 61–76.

Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330. `http://aclweb.org/anthology/J93-2004`.

Martens, Scott (2010). Varro: an algorithm and toolkit for regular structure discovery in treebanks. In *Proceedings of COLING*, pages 810–818. `http://aclweb.org/anthology/C10-2093`.

Moschitti, Alessandro (2006a). Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML*, pages 318–329. Proceedings of ECML.

Moschitti, Alessandro (2006b). Making tree kernels practical for natural language learning. In *Proceedings of EACL*, pages 113–120. `http://aclweb.org/anthology/E06-1015`.

Moschitti, Alessandro, Daniele Pighin, and Roberto Basili (2008). Tree kernels for semantic role labeling. *Computational Linguistics*, 34(2):193–224. `http://aclweb.org/anthology/J08-2003`.

Napoles, Courtney, Matthew Gormley, and Benjamin Van Durme (2012). Annotated gigaword. In *Proceedings of the Joint Workshop on Automatic Knowledge Base Construction and Web-scale Knowledge Extraction*, pages 95–100. `http://aclweb.org/anthology/W12-30`.

O'Donnell, Timothy J., Joshua B. Tenenbaum, and Noah D. Goodman (2009). Fragment grammars: Exploring computation and reuse in language. Technical Report MIT-CSAIL-TR-2009-013, MIT CSAIL. `http://hdl.handle.net/1721.1/44963`.

Post, Matt and Daniel Gildea (2009). Bayesian learning of a tree substitution grammar. In *Proceedings of the ACL-IJCNLP 2009 Conference, Short Papers*, pages 45–48. `http://aclweb.org/anthology/P09-2012`.

Raghavan, Sindhu, Adriana Kovashka, and Raymond Mooney (2010). Authorship attribution using probabilistic context-free grammars. In *Proceedings of ACL*, pages 38–42. `http://aclweb.org/anthology/P10-2008`.

Rieck, Konrad, Tammo Krueger, Ulf Brefeld, and Klaus-Robert Müller (2010). Approximate tree kernels. *Journal of Machine Learning Research*, 11:555–580. `http://jmlr.org/papers/volume11/rieck10a/rieck10a.pdf`.

Sangati, Federico and Willem Zuidema (2011). Accurate parsing with compact tree-substitution grammars: Double-DOP. In *Proceedings of EMNLP*, pages 84–95. `http://aclweb.org/anthology/D11-1008`.

Sangati, Federico, Willem Zuidema, and Rens Bod (2010). Efficiently extract recurring tree fragments from large treebanks. In *Proceedings of LREC*, pages 219–226. `http://dare.uva.nl/record/371504`.

Scha, Remko (1990). Language theory and language technology; competence and performance. In de Kort, Q.A.M. and G.L.J. Leerdam, editors, *Computertoepassingen in de Neerlandistiek*, pages 7–22. LVVN, Almere, the Netherlands. Original title: Taaltheorie en taaltechnologie; competence en performance. Translation available at `http://iaaa.nl/rs/LeerdamE.html`.

Shindo, Hiroyuki, Yusuke Miyao, Akinori Fujino, and Masaaki Nagata (2012). Bayesian symbol-refined tree substitution grammars for syntactic parsing. In *Proceedings of ACL*, pages 440–448. `http://aclweb.org/anthology/P12-1046`.

Skut, Wojciech, Brigitte Krenn, Thorsten Brants, and Hans Uszkoreit (1997). An annotation scheme for free word order languages. In *Proceedings of ANLP*, pages 88–95. `http://aclweb.org/anthology/A97-1014`.

Swanson, Benjamin and Eugene Charniak (2012). Native language detection with tree substitution grammars. In *Proceedings of ACL*, pages 193–197. `http://aclweb.org/anthology/P12-2038`.

van Cranenburgh, Andreas (2012a). Efficient parsing with linear context-free rewriting systems. In *Proceedings of EACL*, pages 460–470. Corrected version: `http://andreasvc.github.io/eacl2012corrected.pdf`.

van Cranenburgh, Andreas (2012b). Extracting tree fragments in linear average time. Technical Report PP-2012-18, FNWI/FGw: Institute for Logic, Language and Computation (ILLC). `http://dare.uva.nl/en/record/421534`.

van Cranenburgh, Andreas (2012c). Literary authorship attribution with phrase-structure fragments. In *Proceedings of the NAACL-HLT 2012 Workshop on Computational Linguistics for Literature*, pages 59–63. Revised version: `http://andreasvc.github.io/clfl2012.pdf`.

van Cranenburgh, Andreas and Rens Bod (2013). Discontinuous parsing with an efficient and accurate DOP model. In *Proceedings of IWPT*, pages 7–16. `http://www.illc.uva.nl/LaCo/CLS/papers/iwpt2013parser_final.pdf`.

van der Beek, Leonoor, Gosse Bouma, Robert Malouf, and Gertjan van Noord (2002). The Alpino dependency treebank. *Language and Computers*, 45(1):8–22.

Zhang, Kaizhong, Rick Statman, and Dennis Shasha (1992). On the editing distance between unordered labeled trees. *Information processing letters*, 42(3):133–139.