Efficiently extract recurring tree fragments from large treebanks

Sangati, F.; Zuidema, W.H.; Bod, L.W.M.

Link to publication

Efficiently extract recurring tree fragments from large treebanks

Federico Sangati, Willem Zuidema, Rens Bod

Institute for Logic, Language and Computation, University of Amsterdam
Science Park 904, 1098 XH Amsterdam, The Netherlands
{f.sangati, zuidema, rens.bod}@uva.nl

Abstract

In this paper we describe `FragmentSeeker`, a tool which is capable to identify all those tree constructions which are recurring multiple times in a large Phrase Structure treebank. The tool is based on an efficient kernel-based dynamic algorithm, which compares every pair of trees of a given treebank and computes the list of fragments which they both share. We describe two different notions of fragments we will use, i.e. standard and partial fragments, and provide the implementation details on how to extract them from a syntactically annotated corpus. We have tested our system on the Penn Wall Street Journal treebank for which we present quantitative and qualitative analysis on the obtained recurring structures, as well as provide empirical time performance. Finally we propose possible ways our tool could contribute to different research fields related to corpus analysis and processing, such as parsing, corpus statistics, annotation guidance, and automatic detection of argument structure.

1. Introduction

In many current linguistic theories, language users produce and understand sentences without necessarily decomposing them to just 'words' and 'rules'; rather, multi-word units may function as the elementary building blocks (Goldberg, 1995; Kay and Fillmore, 1997; Tomasello, 2000). An important question for computational linguistics is whether it can contribute to identifying such building blocks using statistical regularities in large corpora (Culicover and Nowak, 2003; Dennis, 2003; Zuidema, 2006).

A natural assumption in such work is to consider a construction linguistically relevant if there is some empirical evidence about its reusability in a representative corpus of examples. This assumption is at the base of several algorithms in the Data-Oriented-Parsing (DOP) tradition (Bod, 1992; Bod, 2001; Zuidema, 2007) and other approaches based on Tree Substitution Grammars (TSG) (Cohn et al., 2009; O'Donnell et al., 2009). These methods collect a large number of linguistic constructions, viz. *tree fragments*, from a syntactically annotated corpus, in order to parse novel sentences. However, the set of all possible fragments occurring in a big corpus is prohibitively large: it grows exponentially in the size of the treebank trees. Many of the existing approaches are therefore forced to work with a restricted portion of this set, usually a random sample. Moreover, as we will show in section 4.1., the majority of these constructions occur only once in the training corpus, and have little chance to be reused in novel sentences.

So then, how could we extract only those fragments which are reused in the data? In this paper we describe a solution to this challenge. We present `FragmentSeeker`[1], a tool which serves to extract all *recurring maximal fragments*[2], together with their frequencies, from a Phrase Structure (PS) treebank. The tool compares every pair of trees of a given treebank, and computes the list of all maximal fragments which are present in both.

---

[1]The tool is publicly available at `http://staff.science.uva.nl/~fsangati`

[2]All the maximal fragments which occur at least twice in the corpus. For a definition of *maximal fragments*, see section 2.

The tool is based on an efficient kernel-based algorithm, which is conceptually similar to previously proposed methods using this technique (Collins and Duffy, 2001; Moschitti, 2006). The main difference, however, is that, while in previous work kernels are mainly used to numerically quantify the similarity between two trees, in the current project we are interested in identifying the actual constructions they share, i.e. the maximal fragments.

In section 2. we will describe two different notions of fragments which we will use, section 3. will give the details about the implementation, and section 4. will present a case study on the Penn Wall Street Journal (WSJ) treebank (Marcus et al., 1993). Finally section 5. will explain how this methodology could contribute to different research fields related to corpus analysis and processing.

2. Fragments and Partial Fragments

There are two types of constructions that we will focus on. The first one, which we will refer to as *fragment*, is in line with previous work on DOP and TSG. Given a PS tree of a sentence, a fragment extracted from it is a connected subset of nodes, in which every node has either the same daughters as in the original tree, or none.

The second type, which we will call *partial fragment*, is less restrictive, in the sense that it can include any connected subset of nodes of the original tree. This means that for every selected node of a tree structure, any number of its original daughters can be discarded. This types of fragments are particularly interesting if we want to find regularities in flat constructions that present a big number of daughters, e.g. distinguishing the arguments from the adjuncts in a verbal phrase (see also section 5.2.).

Figure 1 shows an example of parsetree, while figure 2 reports one of its fragments (left) and one of its partial fragments (right).

An other notion that we will use in this paper is the one of shared *maximal* (partial) fragment. A (partial) fragment τ shared between to trees is maximal if there is no other shared (partial) fragment starting with the same node as in τ and including all its nodes.
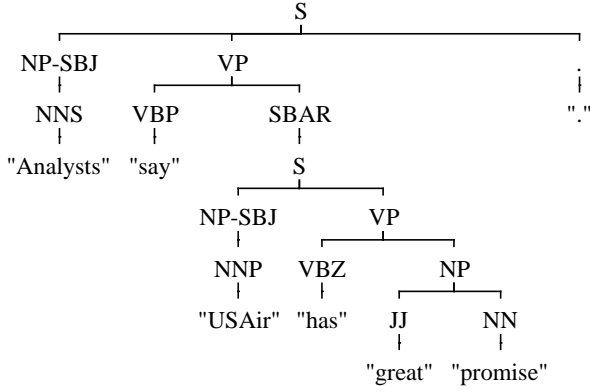
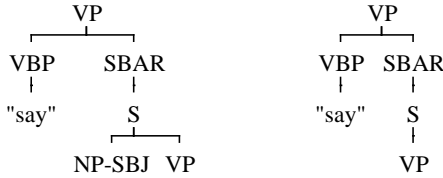Figure 1: Example of a parsetree taken from the WSJ.

Figure 2: Example of a fragments (left) belonging to the tree structure of figure 1, and one of its partial fragments (right). In the fragment, all nodes preserve the same number of daughters as in the original tree, besides the terminal nodes NP-SBJ and VP. In the partial fragment the daughter NP-SBJ of S has been discarded.

3. Implementation Details

In order to extract the fragments and partial fragments which occur at least twice in a given treebank, we compare every pair of trees in the corpus and extract from it all shared maximal (partial) fragments. We employ a kernel method technique previously used in Natural Language Processing (Collins and Duffy, 2001; Moschitti, 2006) to compare the similarity between a pair of trees. While in previous work, the result of the algorithm corresponds to the number of (partial) fragments shared between the two structure, in our implementation we want to keep track of the actual (partial) fragments which are shared.

Algorithm 1 reports the pseudocode of the procedure to extract all maximal fragments and partial fragments from a treebank. For every pair of trees (t_i, t_j) in the corpus, and every pair of nodes (N_i, N_j) between the two structures, algorithm 2 and algorithm 3 are called.

3.1. Maximal Fragments

Algorithm 2 is used to identify the single maximal fragment in common between two PS structures starting at nodes (N_i, N_j). The result is empty in case N_i and N_j differ in their labels, it contains only N_i if the labels are equal but

Algorithm: `ExtractFragments(T)`

Input: a corpus `T` of PS trees
Output: a set of fragments and partial fragments
begin

`FragList`: a set of fragments;
`PartFragList`: a set of partial fragments;
foreach *tree* `t_i` $\in$ `T` **do**
  foreach *tree* `t_j` $\in$ `T` *where* $Ti \neq Tj$ **do**
    foreach *node* `N_i` $\in$ `t_i` **do**
      foreach *node* `N_j` $\in$ `t_j` **do**
        `FragList.addAll(`
          `ExtractMaxFragmenta(N_i,N_j));`
        `PartFragList.addAll(`
          `ExtractMaxPartialFragments(N_i,N_j));`

return {`FragList,PartFragList`};
end

Algorithm 1: Pseudocode of the algorithm used for extracting the list of recurring maximal fragments, and maximal partial fragments from a corpus of PS trees.

Algorithm: `ExtractMaxFragmenta(N_i,N_j)`

Input: two PS nodes (N_i, N_j)
Output: a set of nodes representing the maximal fragment rooted in N_i and N_j
begin

if $N_i \neq N_j$ **then return** {};
`NodeSet` $\leftarrow$ {`N_i`};
if `N_i.daughters` = `N_j.daughters` **then**
  for `d` $\in (1, 2, \ldots, $`N_i.daughters.size`$)$ **do**
    `D_i` $\leftarrow$ `N_i[d]`;
    `D_j` $\leftarrow$ `N_j[d]`;
    `NodeSetD` $\leftarrow$ `ExtractMaxFragmenta(D_i,D_j)`;
    `NodeSet.union(NodeSetD)`;

return `NodeSet`;
end

Algorithm 2: Pseudocode of the algorithm used for extracting the maximal fragments rooted in two nodes of two different PS trees.

their sequences of daughters differ, and it is constituted by multiple nodes if both root labels and daughter sequences coincide. In this last case the result is computed recursively by calling the same function on the pairs of daughters of the two initial nodes occupying the same position.

The time complexity of algorithm 1, when we only extract fragments (and not partial fragments), is $O(n^2 \cdot m^2)$ where n is the size of the treebank and m the number of nodes in the biggest tree of the corpus. In terms of space the number of maximal fragments which are extracted for every pair of trees is in the worst case m^2. In our implementation we make use of *bitset structures* to efficiently represent fragments and perform set operations on them.

Algorithm: `ExtractMaxPartialFragments(N`$_i$`,N`$_j$`)`

Input: two PS nodes (N_i, N_j)
Output: a set of partial fragments rooted in N_i and N_j
begin
 if $N_i \neq N_j$ **then return** $\{\}$;
 MappingsSet $\leftarrow$ MaxSetMappings$(N_i, N_j, 0, 0, \text{true})$;
 if MappingsSet $= \{\}$ **then return** $\{\{N_i\}\}$;
 PartFragSet: a set of partial fragments;
 foreach Mapping $\in$ MappingsSet **do**
  MaxPartialFragmentPairs: an array of sets of
    partial fragments (array size = # pairs in Mapping);
  $i \leftarrow 0$;
  foreach Pair $\in$ Mapping **do**
   $N_1 \leftarrow$ Pair.first;
   $N_2 \leftarrow$ Pair.second;
   MaxPartialFragmentPairs$[i] \leftarrow$
     ExtractMaxPartialFragments(N_1, N_2);
   $i{+}{+}$;
  foreach *way of choosing one set for every element*
   in the array MaxPartialFragmentPairs **do**
   NodeSet $\leftarrow$ union between the chosen sets ;
   NodeSet.union(N_i);
   PartFragSet.*add*(NodeSet);

 return PartFragSet;
end

Algorithm 3: Pseudocode of the algorithm used for extracting the maximal partial fragments rooted in two nodes of two different PS trees.

3.2. Maximal Partial Fragments

Algorithm 3 is used to extract the *maximal partial fragments* shared between two PS structures starting at nodes (N_i, N_j). The main difference with respect to the previous case is that while there is at most one maximal shared fragment starting at a specific pair of nodes of two structures, there might be multiple possible maximal partial fragments in common. In fact, the two nodes might share multiple subsequences of daughters (mappings), but even if they have in common a single subsequence, every pair of corresponding daughters might have in turn multiple subsequences of daughters in common. In this last case every way of choosing a possible tree continuation leads to a different shared maximal partial fragment[3].

At the beginning of algorithm 3 we compute the *Maximal Set of Mappings* (*MSM*) between the two sequences of daughters of the two nodes. This step is performed by algorithm 4, which is a dynamic algorithm similar to the one computing the *Longest Common Subsequence* between two strings (Atallah and Fox, 1998) and the one which finds *All Common Subsequences* (Wang and Lin,

---

[3]This might cause a combinatorial explosion of partial fragments. In our experiments on the WSJ treebank, we found only very few cases where the number of combinations was hard to handle computationally. To solve this issue we have set an upper bound on the number of combinations (1000 by default).

Algorithm: `MaxSetMappings(N`$_i$`,N`$_j$`,x,y,firstCall)`

Input: two PS nodes (N_i, N_j), 2 indexes (x, y) indicating the cell position in the chart table, and a boolean variable (`firstCall`) specifying whether this is the first time the method is being called ;
Output: a set of maximal mapping between the daughters of N_i and N_j
begin
 Mappings: a set of mappings (list of pairs of daughters);
 startX $\leftarrow$ firstCall $?$ x $:$ x $+ 1$;
 startY $\leftarrow$ firstCall $?$ y $:$ y $+ 1$;
 endX $\leftarrow N_i$.daughters.size $- 1$;
 endY $\leftarrow N_j$.daughters.size $- 1$;
 startXExists $\leftarrow$ startX $< N_i$.daughters.size;
 startYExists $\leftarrow$ startY $< N_j$.daughters.size;
 while startXExists $\lor$ startYExists **do**
  if startXExists **then**
   foreach cellY $\in \{\text{endY}, \ldots, \text{startY} + 1\}$ **do**
    if $N_i[\text{startX}] = N_j[\text{cellY}]$ **then**
     endY $\leftarrow$ cellY;
     subMappings $\leftarrow$ MaxSetMappings$(N_i, N_j,$
       startX, cellY, false$)$;
     if $\neg$firstCall **then** add pair $(N_i[\text{startX}],$
       $N_j[\text{cellY}])$ in every mapping of subMappings ;
     Mappings.addAll(subMappings)

  if startYExists **then**
   foreach cellX $\in \{\text{endX}, \ldots, \text{startX} + 1\}$ **do**
    if $N_i[\text{cellX}] = N_j[\text{startY}]$ **then**
     endX $\leftarrow$ cellX;
     subMappings $\leftarrow$ MaxSetMappings$(N_i, N_j,$
       cellX, startY, false$)$;
     if $\neg$firstCall **then** add pair $(N_i[\text{cellX}],$
       $N_j[\text{startY}])$ in every mapping of
     subMappings ;
     Mappings.addAll(subMappings)

  if startXExists $\land$ startYExists $\land$
  $N_i[\text{startX}] = N_j[\text{startY}]$ **then**
   subMappings $\leftarrow$ MaxSetMappings$(N_i, N_j,$
     startX, startY, false$)$;
   if $\neg$firstCall **then** add pair $(N_i[\text{startX}],$
     $N_j[\text{startY}])$ in every mapping of subMappings ;
   Mappings.addAll(subMappings);
   BREAK (while);

  if startX $+ 1 \leq$ endX **then** startX$+{+}$;
  else startXExists $\leftarrow$ false ;
  if startY $+ 1 \leq$ endY **then** startY$+{+}$;
  else startYExists $\leftarrow$ false ;
 return Mappings;
end

Algorithm 4: Pseudocode of the algorithm used for finding the maximal mappings between the daughters of two PS nodes.

	A_0	B_1	B_2	A_3	B_4	B_5
A_0	8			3		
B_1		3	2		1	1
B_2		1	1		1	1

Figure 3: Example of the chart table of algorithm 4, when comparing two nodes P and Q whose set of daughters is $\{A,B,B,A,B,B\}$ and $\{A,B,B\}$ respectively. The number in every cell $c_{i,j}$ corresponds to the total number of maximal mappings starting with the pair $\{P[i],Q[j]\}$. The total number of maximal mappings in this example is 11: 8 starting with the pair $\{A[0],A[0]\}$ and 3 with the pair $\{A[3],A[0]\}$ (see footnote 6). The arrows indicate the sub-mappings contributing to each mapping.

2007). In order to illustrate algorithm 4, figure 3 shows the chart table storing the intermediate results when comparing two nodes P and Q whose sequences of daughters are $\{A_0,B_1,B_2,A_3,B_4,B_5\}$ and $\{A_0,B_1,B_2\}$ respectively[4]. A *mapping* between the two sequences of daughters is a list of pairs of indexes $(p_0,q_0),(p_1,q_1),\ldots,(p_m,q_m)$ such that $P[p_0],P[p_1],\ldots,P[p_m]$ is a subsequence of daughters of P obtained by removing zero or more elements (analogously for Q)[5]. A set of mappings is said to be *maximal* if no mapping in the set is contained in an other mapping. In this example there are in total 11 maximal mappings in $MSM(P,Q)$[6].

The time complexity of algorithm 4 is $O(d^4)$ where d is the maximum number of daughters in the two nodes being analyzed, but the space complexity is exponential in the worse case[7]. In our experiments, we encounter only very few cases when the computation of the MSM becomes intractable[8].

---

[4]The subscript figures are used as placeholders to help distinguishing nodes with the same labels but differing in positions.

[5]This means that the relative positions of the daughters of P and Q is always preserved, and the pairs in a mapping never cross.

[6] $MSM(P,Q) = \{\{(3,0),(4,2)\},\{(3,0),(5,1)\}, \{(3,0),(4,1),(5,2)\},\{(0,0),(1,2)\},\{(0,0),(5,1)\}, \{(0,0),(4,1),(5,2)\},\{(0,0),(2,1),(5,2)\},\{(0,0),(2,1),(4,2)\}, \{(0,0),(1,1),(5,2)\},\{(0,0),(1,1),(4,2)\},\{(0,0),(1,1),(2,2)\}\}$.

[7]For instance if the two sequences of daughters are $\{A_0,A_1,\ldots,A_n\}$ and $\{A_0,A_1,\ldots,A_m\}$, with $n \geq m$, the number of mappings is $n!/((n-m)! \cdot m!)$

[8]These are cases where for instance, a coordinated structure with a big number of conjuncts is paired with a similar but not identical structure. In order to prevent this problem from happening, we generate an approximate solution when the number of MSM, is greater than a specified limit (1000 in the default settings). The approximate solution which is used when the number

3.3. Exact Frequencies

In order to compute the frequencies with which the extracted (partial) fragments occur in the treebank, our algorithm keeps track of the frequency of each encountered shared (partial) fragment. The final count is a very close approximation[9] of the correct one, which nevertheless can be exactly computed by iterating over the tree structures in the corpus, and counting the total number of times each (partial) fragment is present.

4. A case study on the Penn WSJ

In this section we describe some statistics derived from testing `FragmentSeeker` on the Penn WSJ corpus (Marcus et al., 1993). We have restricted the treebank to the 39,832 structures of sections 02-21 (which are typically used as training sections for parsing results), and removed all null productions and traces from the original structures.

4.1. Fragments statistics

Figure 4 reports some statistics on the set of all maximal fragments which are extracted from the treebank. The total number of extracted fragments types is 527,217 (6,179,059 tokens), and their distribution with respect to their depths, reported on the graph of the same figure, shows that fragments of depth 3 and 4 are the most abundant recurring fragments in the corpus.

Figure 5 shows the distribution of the total number of fragments tokens which are present in the same treebank, with respect to their depths and maximum arities. The maximum arity of a fragment corresponds to the maximum number of daughters of the most prolific node. This variable seems to be a primary factor to affect the number of subtrees present in a tree structure.

The total number of fragments without any restriction in depth and arity, is estimated to be $8.7 \cdot 10^{46}$. It follows that the portion of fragment tokens which are recurring in the treebank (shown in the red-colored area in the same graph), is an infinitesimal fraction of all possible fragments[10]. This means that a random uniform sample over the total set of fragments will be likely to miss most if not all recurring fragments.

4.2. Computation time

In terms of empirical computation time, using a 2.53 GHz processor machine, `FragmentSeeker` takes about 0.21 msec to extract fragments from a pair of input tree structures of the treebank (around 46 hours in total), and about 1.13 msec when extracting partial fragments (around 250 hours in total).

---

of MSM exceeds the limit, consists on applying a simple heuristic to extract two mappings, by immediately matching all possible elements and skipping itms only from the first or the second list of daughters in turn. Using the same example, the two mappings in the approximate solution coincide, since no elements need to be skipped: $MSM_{appox}(P,Q) = \{\{(0,0),(1,1),(2,2)\}\}$.

[9]The approximation originates from the fact that we keep track of maximal fragments: if two big structure are identical we extract only one structure for every pair of corresponding nodes.

[10]This fraction is $7.1 \cdot 10^{-41}$. In the graph, the red area doesn't look an infinitesimal fraction simply because the scale on the y-axes is logarithmical.

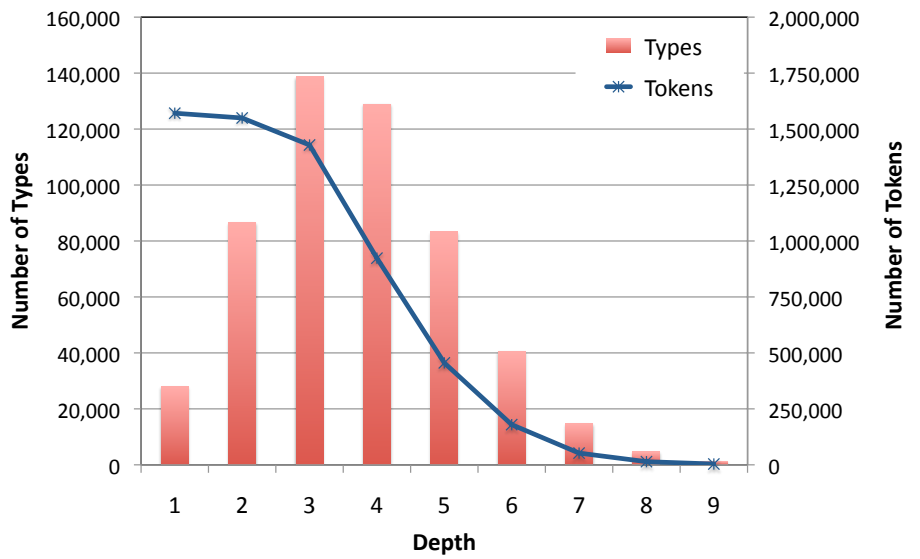| Depth	Types	Tokens
1	27,893	1,570,869
2	86,512	1,549,523
3	138,709	1,428,777
4	128,927	923,315
5	83,218	455,448
6	40,524	179,548
7	14,849	52,424
8	4,677	14,133
9	1,343	3,692
10	398	951
11	96	213
12	39	95
13	14	28
14	6	16
15	4	8
16	3	8
17	2	5
18	2	4
20	1	2
Total	527,217	6,179,059

Figure 4: Distribution of the types and tokens frequencies of the recurring maximal fragment with respect to their depths. Fragments are extracted from sections 02-21 of the Penn WSJ Treebank.
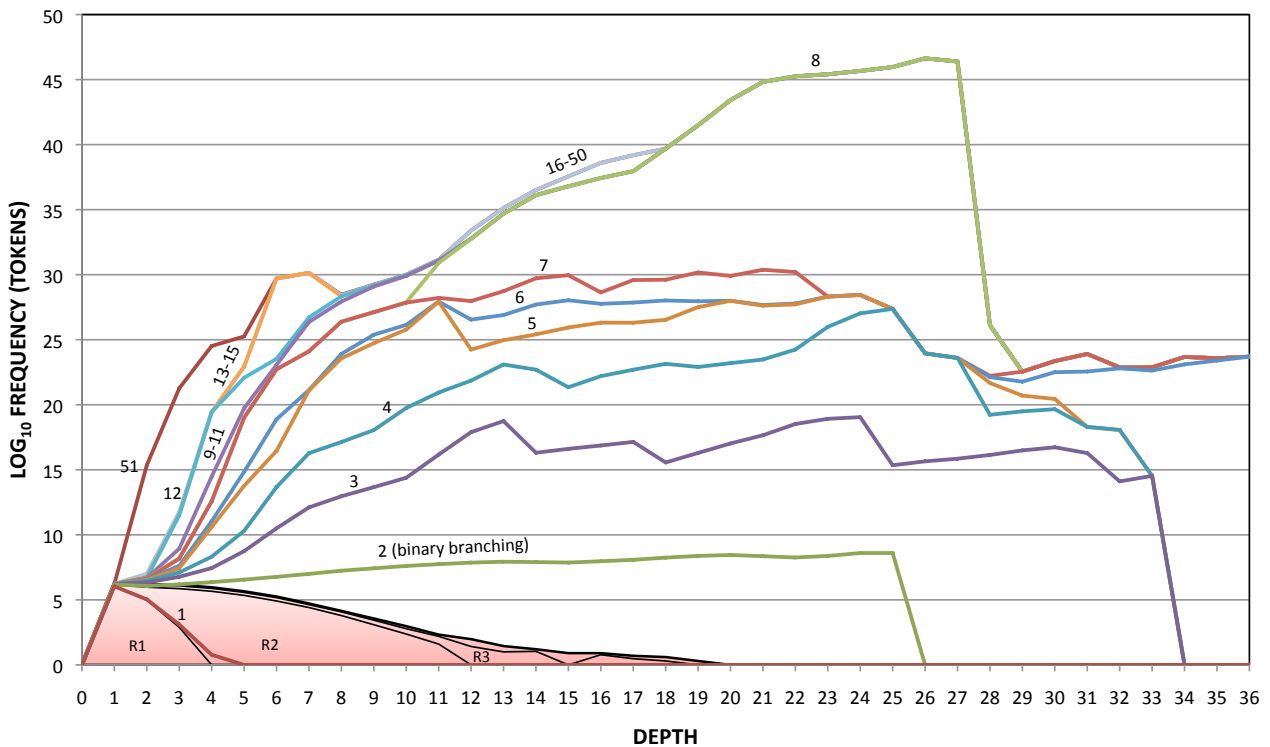

Figure 5: Distribution of the total number of fragments tokens which are present in the trees of sections 02-21 of the Penn WSJ Treebank, with respect to their depths (horizontal axes). Every line corresponds to the total number of fragments at different depth values, when limiting the maximum arity of the fragments to a certain constant (the number reported close to the line). The maximum arity of a fragment is defined to be the maximum number of daughters in the most prolific node of the fragment. The red-colored area at the bottom of the graph represents the portion of fragment tokens which are recurring in the corpus. R1 is the sub-portion including fragments with only unary branching, and similarly R2 and R3 represent sub-portions with fragments with maximum arity 2 and 3.
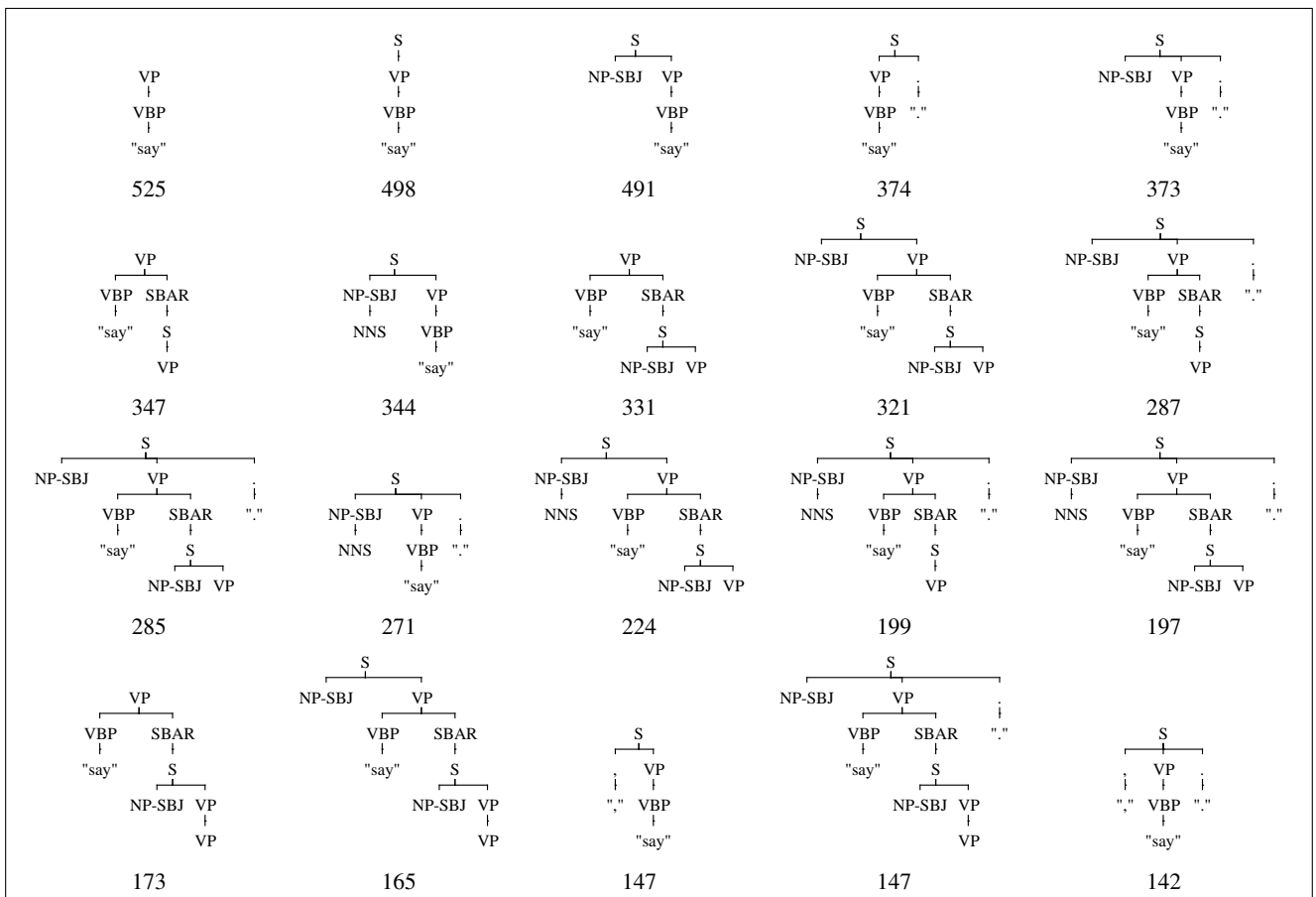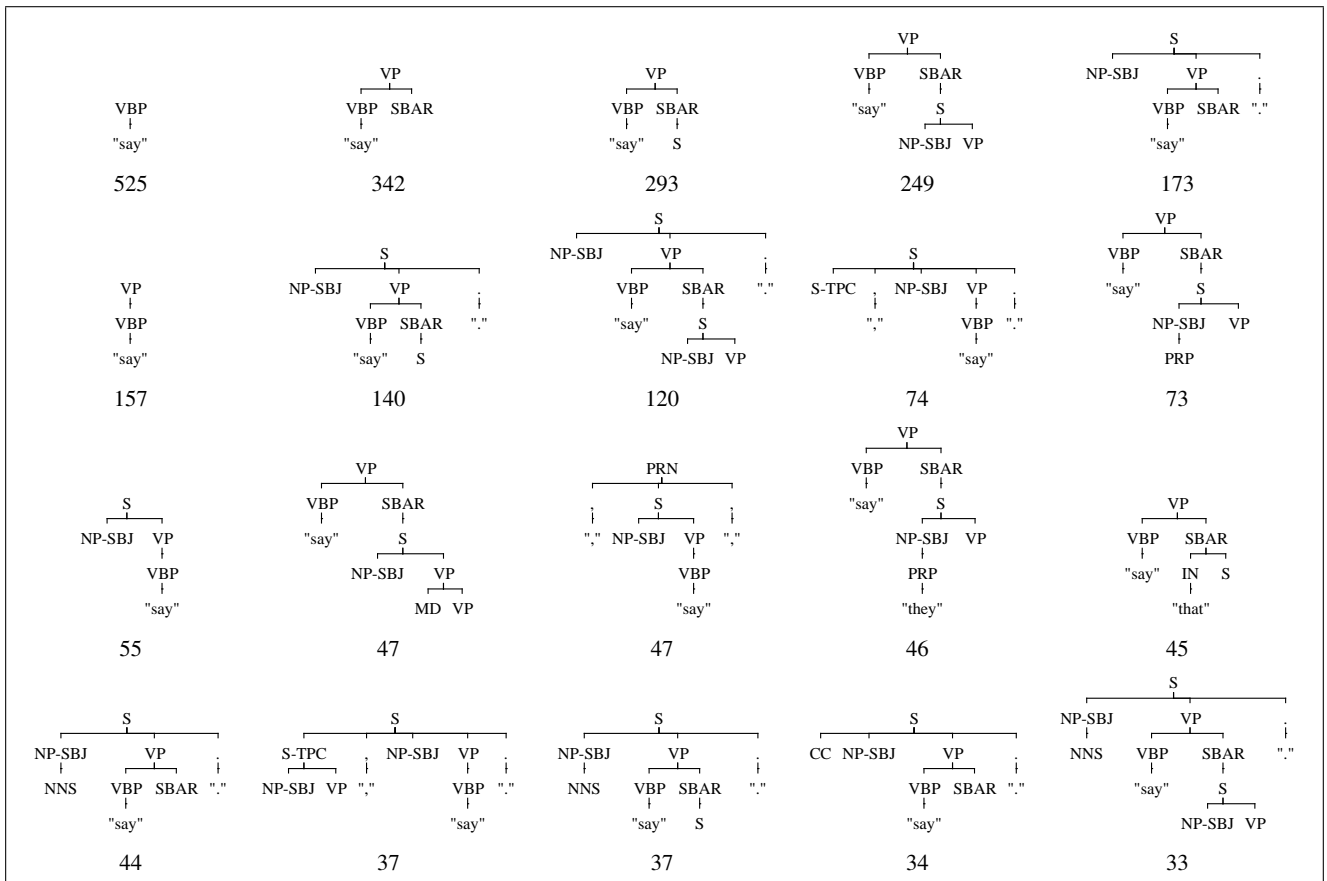
Figure 6: The most frequent fragments (top) and partial fragments (bottom) containing the verb *say*, when it is a present tense verb (VBP). Below each (partial) fragment we report the exact frequency with which it occurs in the WSJ.
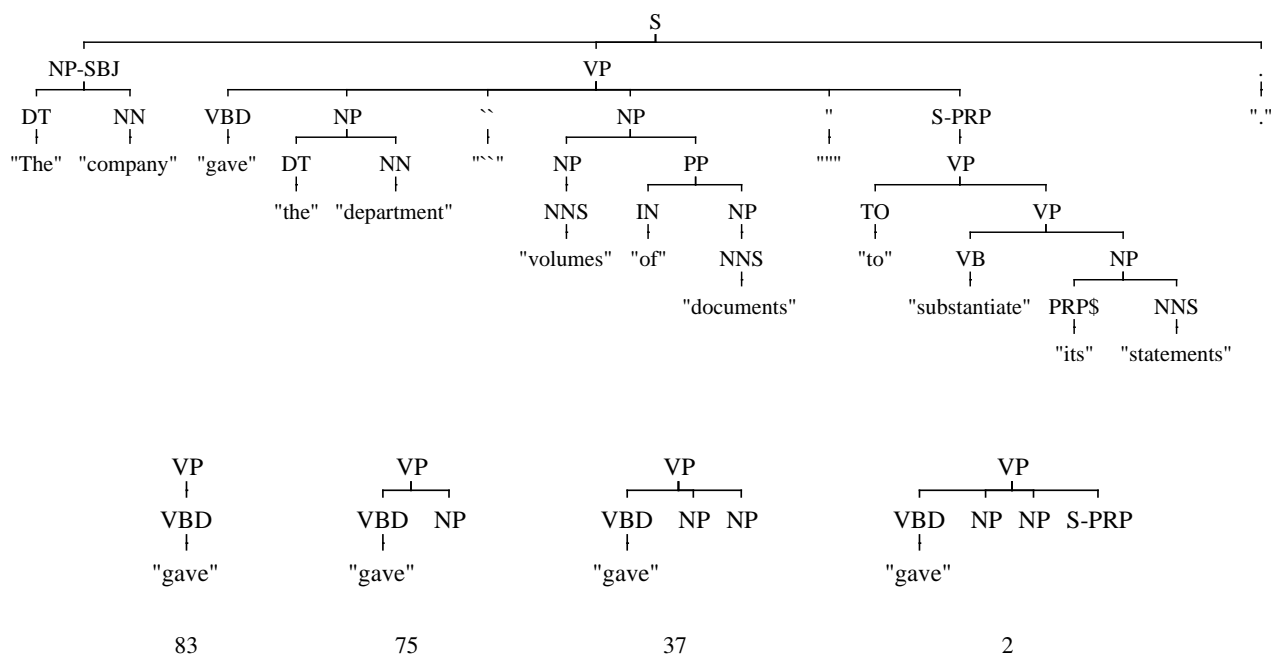
Figure 7: Above: an example of a PS of the WSJ. Below: the most frequent partial fragments present in the PS above, containing the lexical item *gave*.

5. Possible Applications

Annotated corpora are valuable resources for people working with natural languages both from theoretical and practical view points. It is nevertheless not obvious how to approach them when it comes to gather meaningful statistics on the occurrences of specific constructions.

The types of fragments which are extracted from `FragmentSeeker` can easily extract meaningful quantitative and qualitative information from a treebank, which can be used for corpus analysis and processing.

5.1. Corpus analysis

In figure 6 we have reported the most frequent fragments and partial fragments of the WSJ containing the verb "*say*" (when it is a present tense verb). This kind of statistics, can give an insight on the specific template constructions of this particular verb, and can be useful to determine its valency, i.e. the types of arguments it usually accepts. For example, the second fragment at the top of the figure (occurring 342 times) illustrates a specific template construction of the verb, which requires a relative or subordinate clause (SBAR) as first and only argument to its right; this specific construction appears 65% of the times. Moreover the third partial fragment in the second box of the same figure (occurring 491 times) shows that in almost all cases (94%) there is a a subject-noun-phrase (NP-SBJ) preceding the verb.

5.2. Argument/adjunct distinction

The list of partial fragments could also be at the base of an automatic process to infer the argument/adjunct distinction in a corpus of PS trees. This line of research is related to other work on subcategorization (Briscoe and Carroll,

1997) and on the extraction of fragments for more elaborate parsing formalisms such as Tree Adjoining Grammars (Chiang, 2000).

In figure 7 we show an example of a WSJ tree structure which uses the verb *gave*. The frequencies of the partial fragments listed at the bottom of the same figure, give evidence that both NP daughters of the VP are part of the argument structure of the verb. In fact *gave* is used as a ditransitive verb half of the times it appears in the treebank. On the other hand, the last daughter (S-PRP) should be marked as an adjunct, since it is present within the scope of this verb only in an other structure of the treebank (beside the one under consideration).

5.3. Corpus annotation

`FragmentSeeker` could be used as a possible extension of currently available syntactic annotation tools, in projects which aim at expanding partially annotated corpora, as well as those which target a manual revision of automatically annotated treebanks. The availability of the most frequent constructions of a certain lexical item could be beneficial in the annotation and correction process, both in terms of time and consistency.

5.4. Parsing

The set of fragments extracted from a PS treebank together with their frequencies can be easily turned into a probabilistic Tree Substitution Grammar (TSG) for parsing novel sentences (Sangati, 2009). In this framework we can use all the fragments occurring at least twice in the treebank as well as select only those which appear at least a greater number of times. Preliminary results indicate that this direction is promising since it achieves very competitive results even

when the grammar is much smaller than in previous work. The main weakness of TSG is the lack of flexibility of abstracting from the observed local productions (context free grammar rules) in order to be able to generate a subset of daughters from an internal node, or merge several productions into one. We are looking into the possibility of extending the TSG framework to derived a new parsing system based on partial fragments. This would allow to build a more robust and flexible generative grammar without specifying any strong constraint on the types of fragments being used in its derivations as it is the case for Tree Adjoining Grammars (Joshi et al., 1975).

6. Conclusions

We have presented `FragmentSeeker` a kernel-based tool to efficiently extract the most relevant construction from large treebanks. The assumption behind this work is that a particular syntactic construction is relevant if it occurs at least twice in a representative corpus of tree structures. The system is able to select all and only those constructions which recur in a treebank, by iteratively compare every possible pair of structures in the corpus.

The tool is compatible with any phrase structure treebank, and has been tested on the WSJ, where it was successfully used to build a probabilistic grammar for parsing. Furthermore it can be used in different tasks related to corpus analysis and processing, such as guidance tool for syntactic annotators and automatic detection of argument structure.

7. References

Mikhail J. Atallah and Susan Fox, editors. 1998. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA. Produced By Lassandro, Suzanne.

Rens Bod. 1992. A Computational Model Of Language Performance: Data Oriented Parsing. In *COLING*, pages 855–859.

Rens Bod. 2001. What is the minimal set of fragments that achieves maximal parse accuracy? In *ACL*, pages 66–73.

Ted Briscoe and John Carroll. 1997. Automatic extraction of subcategorization from corpora. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 356–363, Washington, DC, USA, March. Association for Computational Linguistics.

David Chiang. 2000. Statistical parsing with an automatically-extracted tree adjoining grammar. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 456–463, Morristown, NJ, USA. Association for Computational Linguistics.

Trevor Cohn, Sharon Goldwater, and Phil Blunsom. 2009. Inducing Compact but Accurate Tree-Substitution Grammars. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 548–556, Boulder, Colorado, June. Association for Computational Linguistics.

Michael Collins and Nigel Duffy. 2001. Convolution Kernels for Natural Language. In Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, editors, *NIPS*, pages 625–632. MIT Press.

Peter W. Culicover and Andrzej Nowak. 2003. *Dynamical Grammar*. Oxford University Press.

Simon Dennis. 2003. A comparison of statistical models for the extraction of lexical information from text corpora. In *Proceedings 25th Annual Meeting of the Cognitive Science Society (CogSci 2003), Boston, USA, 31 July 2 August*.

A.E. Goldberg. 1995. *Constructions: A Construction Grammar Approach to Argument Structure*. University Of Chicago Press.

Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. 1975. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163.

Paul Kay and Charles J. Fillmore. 1997. Grammatical constructions and linguistic generalizations: the what's x doing y? construction. *Language*, 75:1–33.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Alessandro Moschitti. 2006. Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees. In *ECML*, pages 318–329, Berlin, Germany, September. Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings.

Timothy J. O'Donnell, Noah D. Goodman, and Joshua B. Tenenbaum. 2009. Fragment Grammars: Exploring Computation and Reuse in Language. Technical Report MIT-CSAIL-TR-2009-013, MIT.

Federico Sangati. 2009. A simple dop model for constituency parsing of italian sentences. In *Proceedings of EVALITA 2009*.

Michael Tomasello. 2000. The item-based nature of children's early syntactic development. *Trends in cognitive sciences*, 4:156–163.

Hui Wang and Zhiwei Lin. 2007. A novel algorithm for counting all common subsequences. *Granular Computing, IEEE International Conference on*, 0:502.

Willem Zuidema. 2006. What are the productive units of natural language grammar?: a DOP approach to the automatic identification of constructions. In *CoNLL-X '06: Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 29–36, Morristown, NJ, USA. Association for Computational Linguistics.

Willem Zuidema. 2007. Parsimonious Data-Oriented Parsing. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 551–560, Prague, Czech Republic, June. Association for Computational Linguistics.