Quantitative analysis of treebanks using frequent subtree mining methods

Scott Martens
Centrum voor Computerlinguïstiek, KU Leuven
Blijde-Inkomststraat 13, bus 3315
3000 Leuven Belgium
scott@ccl.kuleuven.be

Abstract

The first task of statistical computational linguistics, or any other type of data-driven processing of language, is the extraction of counts and distributions of phenomena. This is much more difficult for the type of complex structured data found in treebanks and in corpora with sophisticated annotation than for tokenized texts. Recent developments in data mining, particularly in the extraction of frequent subtrees from treebanks, offer some solutions. We have applied a modified version of the *TreeMiner* algorithm to a small treebank and present some promising results.

1 Introduction

Statistical corpus linguistics and many natural language processing applications rely on extracting the frequencies and distributions of phenomena from natural language data sources. This is relatively simple when language data is treated as bags of tokens or as n-grams, but much more complicated for corpora annotated with complex feature schemes and for treebanks where syntactic dependencies are marked. A great deal of useful information is encoded in these more complex structured corpora, but access to it is very limited using the traditional algorithms and analytical tools of computational linguistics. Many of the most powerful techniques available to natural language processing have been built on the basis of *n-gram* and *bag of words* models, but we already know that these methods are inadequate to fully model the information in texts or we would have little use for treebanks or annotation schemes.

Suffix trees provide some improvement over n-grams and bag-of-words schemes by identifying all frequently occurring sequences regardless of length (Weiner, 1973; McCreight, 1976; Ravichandran and Hovy, 2002). While this has value in identifying some multi-word phenomena, any algorithm that models languages on the basis of frequent contiguous string discovery will have trouble modeling a number of pervasive phenomena in natural language. In particular:

- Long distance dependencies – i.e., dependencies between words that are too far apart to be accessible to n-gram models.
- Flexible word orders – languages usually have contexts where word order can vary.
- Languages with very rich morphologies that *must* be taken into account or where too much important information is lost through lemmatization.
- Correlations between different levels of abstraction in annotation, such as between the lemma of a verb and the semantic or syntactic class of its arguments.
- Extra-syntactic correlations that may involve *any* nearby word, such as semantic priming effects.

In treebanks and other annotated corpora that can be converted into rooted, directed graphs, many of these phenomena are accessible as *frequently recurring subtrees*. For example, consider the Dutch idiom “naar huis gaan”, (*to go home*). The components of this phrase can appear in a variety of orders and with words inserted between the constituents:

1. Ik zou naar huis kunnen gaan. (I could go home.)
2. We gaan naar huis. (We’re going home.)

In a treebank, these two sentences would share a common subtree that encompasses the phrase “naar huis gaan”, as in Figure 1. Note that for this purpose, two subtrees are treated as identical if the

Proceedings of the 2009 Workshop on Graph-based Methods for Natural Language Processing, ACL-IJCNLP 2009, pages 84–92,
Suntec, Singapore, 7 August 2009.

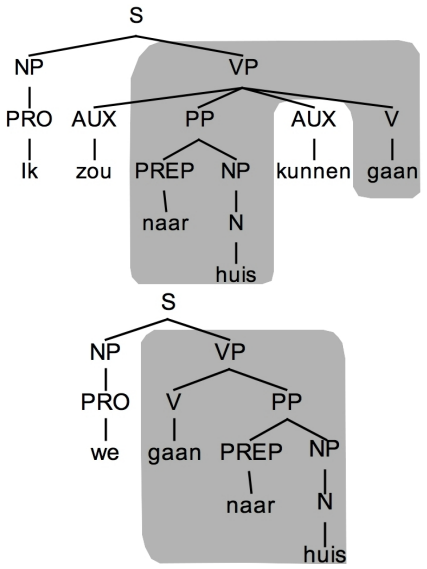


Figure 1: The two Dutch sentences *Ik zou naar huis kunnen gaan* and *We gaan naar huis*, parsed, and with the frequent section highlighted. Note that these two subtrees are identical except for the order of the nodes. (N.B.: This tree does not take the difference between the infinitive and conjugated forms into account.)

only difference between them is the order of the children of some or all the nodes.

Most theories of syntax use trees to represent interlexical dependencies, and generally theories of morphology and phonology use either hierarchical tree structures to represent their formalisms, or use unstructured bags that can be trivially represented as trees. Most types of linguistic feature systems are at least in part hierarchical and representable in tree form. Because so many linguistic phenomena are manifest as frequent subtrees within hierarchical representations that are motivated by linguistic theories, efficient methods for extracting frequent subtrees from treebanks are therefore potentially very valuable to corpus and computational linguistics.

2 Previous and Related Work

Tree mining research is a subset of graph mining focused specifically on rooted, directed acyclic graphs. Although there is research into extracting frequent subtrees from free (unrooted and undirected) trees, free tree mining usually proceeds by deciding which node in any particular tree will be treated as a root, and then treating it as if it was a rooted and directed tree (Chi et al., 2003).

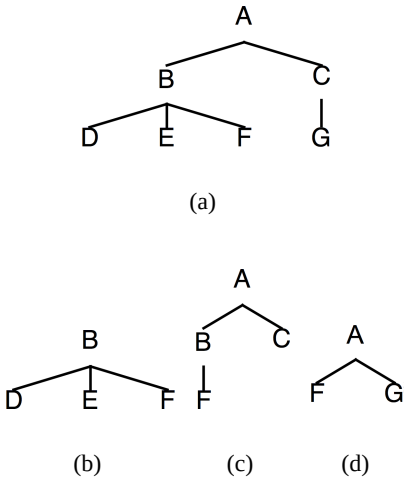


Figure 2: Tree (a) and different types of subtree: (b) a *bottom-up* subtree of (a), (c) an *induced* subtree of (a), and (d) an *embedded* subtree of (a).

Research on frequent subtree discovery generally draws heavily on early work by Zaki (2002) and Asai et al. (2002) who roughly simultaneously began applying the *Apriori* algorithm to frequent tree discovery (Agrawal et al., 1993). For a summary of *Apriori*, which is widely used in data mining, and a short review of its extensive literature, see Kotsiantis and Kanellopoulos (2006). A broad summary of algorithms for frequent subtree mining can be found in Chi et al. (2004).

Research into frequent substructures in computational linguistics is quite limited. The *Data Oriented Processing* model (Bod et al., 2003) along with its extension into machine translation - the *Data Oriented Translation* model (Poutsma, 2000; Poutsma, 2003; Hearne and Way, 2003) - is the most developed approach to using frequent subtree statistics to do natural language processing. There is also growing work, largely stemming out of DOP research, into subtree alignment in bilingual parsed treebanks as an aid in the development of statistical and example-based machine translations systems (Hassan et al., 2006; Tinsley et al., 2007; Zhechev and Way, 2008).

3 Key concepts

Among the key concepts in tree mining is the difference between *bottom-up subtrees*, *induced subtrees* and *embedded subtrees*. A *bottom-up* subtree T' of a tree T is a subtree where, for every node in T', if its corresponding node in T has children, then all those children are also in T'. An *induced*

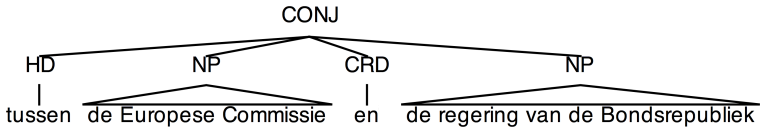


Figure 3: *"...between the European Commission and the government of the [German] Federal Republic..."* This structure is a subtree of one of the sentences in the Alpino corpus of Dutch where a node has two children with the same labels - two NPs. This often occurs with conjunctions and can prevent the algorithm from discovering some frequent subtrees.

subtree T' of T is a subtree where every node in T' is either the root of T' or its parent in T is also its parent in T'. An *embedded* subtree T' of T is a subtree where every node in T' is either the root of T' or its parent in T' is one of its ancestors in T. See Figure 2 for an example of these different types of subtrees.

Linear time solutions exist for finding all frequent *bottom-up* subtrees in a treebank because this problem can be transformed into finding all frequent substrings in a string, a problem for which fast solutions are well known (Luccio et al., 2001; Luccio et al., 2004).

Solutions for *induced* and *embedded* subtrees draw heavily on Zaki (2002) (the *TreeMiner* algorithm) and Asai et al. (2002) (the *FREQT* algorithm), both of whom propose *Apriori*-style approaches. This type of solution has the general property that runtime is proportionate to the size of the *output*: the sum of the number of times each frequent subtree appears in the treebank. This is not readily predictable, because the number and frequencies of subtrees is not formally determinable from the size of the treebank and can grow very rapidly.

3.1 Ordered and unordered trees

TreeMiner/*FREQT* approaches require all trees to be *ordered* so that the nodes of any frequent subtree will always appear in the same order every time it appears. The children of each non-leaf node are sorted into a lexicographic order, *but this only guarantees that frequent subtrees will always appear with the same ordering if no node has more than one non-leaf child node with the same label*. This is not uniformly true of natural language parse trees, as shown in Figure 3. Solutions exist that remove this limitation - notably Chi et al. (2003) - but they come at a significantly increased processing cost.

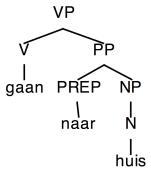
3.2 Closed trees

Given that this type of approach to subtree discovery has runtime bounds proportionate to the unpredictable size of the output, one way to keep subtree discovery within manageable bounds is to restrict the output. Many of the frequent trees present in treebanks are *redundant*, since they are identically distributed with other, larger trees, as in Figure 4.

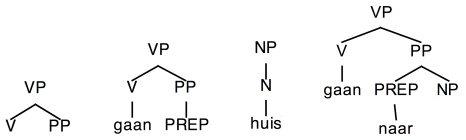
If a corpus has a sequence of tokens $ABCDE$ that appears f times, then that corpus also contains at least f instances of the sequences A, B, C, D, E, AB, BC, CD, DE, ABC, BCD, CDE, $ABCD$, and $BCDE$. If any of these sequences appears *only* in the context of $ABCDE$, then they are *redundant*, because they have the same count and distribution as the longer sequence $ABCDE$.

If a set of sequences is *identically distributed* - appearing in all the same places - then the longest of those sequences is called a *closed sequence*. In more formal terms, a sequence S that appears f times in a corpus is called closed if and only if there is no prefix or suffix a such that aS or Sa also appears f times in the corpus. This definition extends easily to trees: A subtree T in a treebank is closed if and only if there is no node that can be added to it to produce a new subtree T' such that the frequency of T' is equal to the frequency of T. All subtrees in a corpus are either closed subtrees or are subtrees of closed subtrees that appear in exactly the same places in the treebank. The set of closed subtrees in a treebank is the smallest set of subtrees that encompasses all the distributions of subtrees in the treebank. Any subtree that is not in the list of closed subtrees is either a subtree of one of the closed subtrees that appears exactly as often and in all the same places, or does not appear in the treebank at all.

There are algorithms that extract *only* closed subtrees from treebanks - notably Chi et al. (2005a) - and thereby increase their speed dramatically without producing less



(a) The common subtree of the two parse trees in Figure 1: "naar huis gaan"



(b) Redundant subtrees of tree (a). There are many more such structures.

Figure 4: Closed and non-closed subtrees in *just* the two sentences in Figure 1. In a larger treebank, some of these might not be redundant.

information, since any non-closed subtree present in the treebank is a subtree of a closed one and shares its distribution.

4 Algorithm and data structures

The algorithm used in this research is an extension of the *TreeMiner* algorithm (Zaki, 2002), modified to extract only closed subtrees. It takes a minimum frequency threshold as a parameter and extracts only those subtrees which are closed and whose frequency is at least equal to the threshold. This algorithm suffers from the same shortcoming of Zaki's original algorithm in that it is only guaranteed to find all frequent subtrees among *ordered* trees where no node has two non-leaf children with the same label.

It has one novel property which it appears not to share with any other subtree extraction scheme to date: This algorithm outputs subtrees in order from the most frequent to the least. Given the difficulty of predicting in advance how large the output will be, and the large size of many natural language data sources, this can be a real boon. If output size or memory usage grow too large, or too much time has passed, the program can be stopped while still guaranteeing that it has not missed any more frequent subtree than the last one outputted.

This section can only very briefly describe the algorithm.

4.1 Definitions

A treebank is any collection of trees where each node bears a label and each node is uniquely addressable in such a way that the address a_n of a node n is always greater than the address a_p of its parent p. This is accomplished by representing all trees as *ordered depth-first canonical strings*. (See Chi et al. (2005b).)

Each appearance of a subtree within a treebank is characterized by the address of its root in the treebank and the address of its rightmost node. This data structure will be called a *Hit*. The list of all *Hits* corresponding to all the appearances of some subtree in the treebank will be called a *HitList*. So, for each subtree there is a corresponding *HitList* and vice-versa. *HitLists* are always constructed in sequential order, from first instance in the treebank to last, and can never contain duplicates.

We will define the function *queueKey* on *HitLists* to output an array of four numbers in a specific order, given a *HitList* as input:

1. The number of *Hits* in the *HitList*.
2. The distance from the address of the root of the first *Hit* to the *end* of the treebank.
3. The distance from the address of the rightmost node of the first *Hit* to the end of the treebank.
4. The number of nodes in the subtree associated with that *HitList*.

These keys are sortable and designed to ensure that *HitLists* from a single treebank can always be sorted into a fixed order such that, for two *HitLists* A and B, if $A > B$ then:

1. A has more *Hits* than B.
2. If A has the same number of *Hits* as B, then the root of the first *Hit* in A precedes the root of the first *Hit* in B.
3. If A's first root is identical to B's, then the address of the rightmost node of the first *Hit* in A precedes the address of the rightmost node of the first *Hit* in B.
4. If the first *Hit* in A is exactly the same the first *Hit* in B, then the subtree associated with A has more nodes than the subtree associated with B.

A *self-sorting queue* is any data structure that stores key-data pairs and stores the keys in order from greatest to least. The data structure used to implement a self-sorting queue in this research is an *AVL tree* (Adelson-Velskii and Landis, 1962), however, other structures could equally well have been used. The self-sorting queue will be used to maintain a sorted list of *HitLists*, sorted in the order of their *queueKeys* as described above.

4.2 Initialization

Fix a minimum frequency threshold t for the subtrees you wish to extract from the treebank. Start processing by initializing one *HitList* for each unique label in the treebank with the set of *Hits* that corresponds to each occurrence of that label. We will treat each as a *HitList* with an associated subtree containing only one node. This set is constructed in linear time by iterating over all the nodes in the treebank.

Of the initial *HitLists*, throw away all those with fewer than threshold frequency t *Hits* in them. The remaining *HitLists* are inserted into the self-sorting queue.

4.3 Extracting induced subtrees without checking for closure

Extracting *all* the subtrees above a fixed frequency - not just the closed subtrees - in order from the most frequent to the least, proceeds as follows:

1. **Initialize** as described in Section 4.2.
2. **Pop** the top *HitList* hl and its associated subtree s from the queue.
3. **Extend** hl:
   (a) **Visit** each *Hit* in hl and find all the nodes that can be added to the right side of s to produce new induced subtrees.
   (b) **Generate** new *HitLists* for all subtrees that extend s by one node to the right.
   (c) **Test** each new *HitList* to make sure it appears more than threshold frequency t times, and if it does, insert it into the queue.
   (d) **Output** s and hl.
4. **Repeat** until the queue is empty.

This is essentially identical to the *TreeMiner* and *FREQT* algorithms already published by Zaki (2002) and by Asai et al. (2002), except that it outputs frequent subtrees in order from the most frequent to the least.

4.4 Extracting only closed induced subtrees

By controlling the order in which *HitLists* reach the top of the queue, it is possible to efficiently prevent any subtree which is not a closed subtree or a prefix of a closed subtree from being extended, and to prevent any subtree that is not closed from being outputted.

Every subtree with a frequency of f is either a closed subtree, a prefix of a closed subtree that also has a frequency of f and can be constructed by adding more nodes to the right, or is a redundant non-closed subtree that need not be extended or stored. Consider a redundant, non-closed subtree x and a closed subtree or prefix of a closed subtree y which has the same frequency, and has the same set of addresses for the rightmost node of each of its appearances in the treebank. The sort order of the self-sorting queue (see Section 4.1) ensures that if a prefix of a closed subtree y is in the queue and some subtree of it x is also in the queue, then y is closer to the top of the queue than x is. Furthermore, it can be proven that the prefix of a closed subtree with the same distribution as any non-closed, redundant subtree will be generated, inserted into the queue, and removed from the top of the queue before x can reach the top.

So, to prevent x from being extended or stored, all that is necessary is to check to see there is some closed subtree or prefix of a closed subtree y such that:

- y has already been at the top of the queue.
- y has the same frequency as x.
- The set of rightmost nodes of every *Hit* in y's *HitList* is identical to the set of rightmost nodes of every *Hit* in x's *HitList*.
- x is a subtree of y

This can be checked by constructing a hash value for each *HitList* based on its frequency and some subset of the set of rightmost nodes of every *Hit*. In our experiments, we used only the first node of each HitList. If x's hash value matches some previously processed y's hash value, then check if x is a subtree of y and reject it if it is. The result is to only instantiate closed subtrees and their prefixes, and subtrees which are one node extensions of closed subtrees and their prefixes.

Like *TreeMiner*, worst case space and time bounds are proportionate to the number of subtrees instantiated and the number of times each appears in the corpus. This is smaller than the

worst case bounds for *TreeMiner* because it does not instantiate all frequent subtrees. There is additional approximately constant time processing for each instantiated subtree to check for closure and to store it in the self-sorting queue. At the lowest frequency thresholds, this can take up the majority of runtime, but is generally negligible at high frequencies.

5 Results

We applied this algorithm to a parsed and hand-corrected 7137 sentence subset of the Alpino Treebank of Dutch.[1] The average sentence length in this small treebank is roughly 20 words, and the corresponding trees have an average of approximately 32 nodes for a total of 230,673 nodes. With the minimum frequency set to 2, this algorithm extracted 342,401 closed subtrees in about 2000 seconds on a conventional workstation running Linux[2]. The same implementation but without testing for closure - which makes this algorithm equivalent to *TreeMiner* - extracted some 4.2 million trees in roughly 11,000 seconds. Closed tree extraction contrasts quite favorably to extraction without closure, even over a small dataset.

Min. Freq. Threshold	Subtrees extracted	Runtime
2	342401	1952.33s
3	243484	1004.30s
4	176885	588.58s
5	134495	402.26s
8	72732	209.51s
10	53842	163.22s
15	30681	112.39s
20	20610	85.24s
30	11516	66.05s
40	7620	54.14s
50	5549	47.98s
60	4219	43.24s
70	3365	39.97s

Table 1: Runtime and closed trees extracted at different minimum frequency thresholds, using the 7137 sentence sample of the Alpino Treebank.

Runtime and the number of trees produced fall very dramatically as thresholds rise - so much so that setting the minimum frequency to 3 instead of 2 *halved* the runtime. This pattern is characteristic of a power law distribution like Zipf's law. (See Table 1 and Figure 5.) Given the pervasiveness of power law distributions in word frequencies, it should perhaps not be surprising to discover that frequent closed subtrees in treebanks are similarly distributed. This research may be the first effort to empirically support such a conclusion, although admittedly only very tentatively.

Sentences	Total nodes	Subtrees extracted	Runtime
2500	94528	37607	61.08s
5000	189170	98538	260.91s
10000	379980	264616	1495.19s
15000	573629	477750	3829.29s
20000	763502	704018	7998.57s

Table 2: Runtime and closed trees extracted from automatically parsed samples of the *Europarl* Dutch corpus, keeping the minimum frequency threshold constant at 5 for all sizes of treebank.

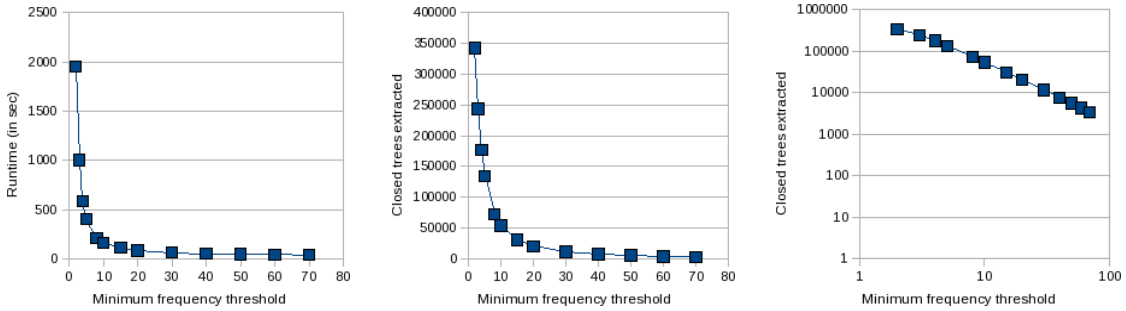
To test the impact of varying the size of the treebank, but keeping the minimum frequency threshold constant, we used a section of the Dutch portion of the *Europarl corpus* (Koehn, 2005) automatically parsed using the Alpino Dutch parser (van Noord, 2006) without any manual correction. Random samples of 2500, 5000, 10000, 15000 and 20000 sentences were selected, and all subtrees of frequency 5 or higher were extracted from each, as summarized in Table 2. As treebank size grows, the number of subtrees extracted at the same minimum frequency threshold, and the time and memory used extracting them, grows exponentially. This is in sharp contrast to algorithms that extract frequently recurring strings, which increase linearly in time and memory usage as the data grows.

However, if the minimum frequency threshold is kept constant as a proportion of the size of the treebank, then the number of trees extracted remains roughly constant and the time and memory used to extract them grows roughly linearly with the size of the treebank. Table 3 shows the result for different sized random samples of the parsed *Europarl* corpus.

Lastly, since this algorithm has known difficulties when presented with trees where more than one non-leaf child of a node can have the same

[1] `http://www.let.rug.nl/vannoord/trees/`

[2] A Dell Precision 490 workstation with an Intel Dual-Core Xeon processor and 8GB of memory. The algorithm was not implemented to use two processors.



(a) Runtime by minimum frequency threshold.

(b) Subtrees extracted by minimum frequency threshold.

(c) Log-log plot of (b).

Figure 5: Runtime (a) and subtrees extracted (b) from the Alpino sample using different minimum frequency thresholds. Figure (c) is a log-log plot of (b). Figure (c) looks close to a straight line, which is characteristic of a power law distribution.

Sents	Total nodes	Min. Freq. Thres.	Subtrees extracted	Run time
2500	99323	5	42905	72.95s
5000	194736	10	42783	122.18s
10000	382022	20	41988	216.23s
15000	574632	30	43078	325.86s
20000	770240	40	44416	435.19s

Table 3: Runtime and closed trees extracted from automatically parsed samples of the *Europarl* Dutch corpus, with minimum frequency thresholds kept roughly constant as a proportion of the sample size.

Min. Freq. Threshold	Non-closed trees	as a % of all trees extracted
2	2874	0.84%
3	693	0.28%
4	225	0.13%
5	101	0.08%
8	18	0.02%
10	11	0.02%
15	6	0.02%
20	3	0.01%
30	0	0.00%

Table 4: Non-closed trees from the 7137 sentence sample of the Alpino Treebank, produced erroneously as closed trees because of repeated labels. There were no non-closed trees extracted at frequencies over 30.

label (see sections 3.1 and 4), we attempted to determine if this problem is marginal or pervasive. The 7137 sentence Alpino Treebank sample contains 3833 nodes with more than one non-leaf child node with identical labels or roughly 1.7% of all nodes. Furthermore, these nodes are present in 2666 sentences - some 37% of all sentences! This is a very large minority.

In order to estimate the effect this phenomenon has on the extraction of closed trees, we looked for outputted trees that are not closed by comparing the *HitLists* of all outputted trees to all other outputted trees with the same frequency. Table 4 shows the number of trees with identical distributions to other outputted trees - i.e. trees that appeared to be closed to this algorithm, but in fact are not. The number was surprisingly large, but distributed overwhelmingly at the very lowest frequencies.

6 Conclusions

The algorithm presented above opens up treebanks and annotated corpora to much more detailed quantitative analysis, and extends the tools available for data-driven natural language processing. This makes a number of new applications possible. We are developing treebank indexing for fast retrieval by tree similarity, in order to make full treebanks available for example-based parsing and machine translation in real time. This algorithm also has applications in constructing concordances of syntactic, morphological and semantic structures - types of information that are not traditionally amenable to indexing. Furthermore, statistical models of natural language data can take

advantage of comprehensive subtree censuses to become fully syntax-aware, instead of relying on *bag of words* and *n-gram* models.

However, there are a number of drawbacks and caveats that must be highlighted.

Runtime, memory usage and output size are difficult to estimate in advance. This is mitigated in part by the order in which subtrees are outputted, making it possible to extract only the most frequent subset of subtrees given fixed time and space bounds. Empirically, it appears that resource requirements and output size can also be estimated by sampling, if minimum frequency thresholds can be kept constant as a proportion of total treebank size.

The formalisms used in most theories of syntax allow nodes to have multiple non-leaf children with the same labels. Although errors caused by non-unique labels are overwhelmingly present only among the lowest frequency subtrees, errors appear often enough to pose a non-negligible problem for this algorithm.

We are investigating the degree to which this can be mitigated by making different choices of linguistic formalism. Syntax trees that contain only binary trees - for example, those constructed using *Chomsky Normal Form* rules (Jurafsky and Martin, 2009) - cannot have identically labelled non-leaf children, but must suffer some loss of generality in their frequent subtrees because if it. Other theories can reduce the size of this source of error, notably dependency syntax which often uses fewer abstract labels (Tesnière, 1959; Mel'čuk, 1988; Sugayama and Hudson, 2006), but will most likely be poor sources of highly general rules as a consequence.

Furthermore, tree mining algorithms exist that eliminate this problem, but at some cost. We are investigating a hybrid solution to the non-unique label problem that identifies only those subtrees where more resource-intensive closure checking is necessary. This will guarantee the correct extraction of closed subtrees in all cases while minimizing the additional processing burden.

Among the open problems suggested by this research is the degree to which the empirical results obtained above are dependent on the language of the underlying data and the linguistic formalisms used to produce treebanks. Different linguistic theories use different abstractions and use their abstract categories differently. This has an immediate effect on the number of nodes in a treebank and on the topology of the trees. Some theories produce more compact trees than others. Some produce deep trees, others produce shallow trees. It is likely that the formalisms used in treebanks have a pervasive influence on the number and kind of frequent subtrees extracted. By doing quantitative research on the structures found in treebanks, it may become possible to make reliable operational choices about the linguistic formalisms used in treebanks on the basis of the kinds of structures one hopes to get out of them.

Acknowledgments

This research is supported by the AMASS++ Project[3] directly funded by the *Institute for the Promotion of Innovation by Science and Technology in Flanders* (*IWT*) (SBO IWT 060051).

References

Georgiy M. Adelson-Velskii and Yevgeniy M. Landis. 1962. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146(2):263–266.

Rakesh Agrawal, Tomasz Imielinski and Arun Swami. 1993. Mining association rules between sets of items in large databases. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 207–216.

Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto and Setsuo Arikawa. 2002. Efficient substructure discovery from large semi-structured data. *Proceedings of the Second SIAM International Conference on Data Mining*, 158–174.

Rens Bod, Khalil Sima'an and Remko Scha, editors. 2003. *Data-Oriented Parsing*. CLSI Publicatons, Stanford, CA.

Yun Chi, Yirong Yang and Richard R. Muntz. 2003. Indexing and Mining Free Trees. *UCLA Computer Science Department Technical Report No. 030041*.

Yun Chi, Richard R. Muntz, Siegfried Nijssen and Joost N. Kok. 2004. Frequent Subtree Mining – An Overview. *Fundamenta Informaticae*, 66(1-2):161–198.

Yun Chi, Yi Xia, Yirong Yang and Richard R. Muntz. 2005a. Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):190–202.

[3] `http://www.cs.kuleuven.be/~liir/projects/amass/`

Yun Chi, Yi Xia, Yirong Yang and Richard R. Muntz. 2005b. Canonical forms for labelled trees and their applications in frequent subtree mining. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):203–234.

Hany Hassan, Mary Hearne, Khalil Sima'an and Andy Way. 2006. Syntactic Phrase-Based Statistical Machine Translation. *Proceedings of the IEEE 2006 Workshop on Spoken Language Translation*, 238–241.

Mary Hearne and Andy Way. 2003. Seeing the Wood for the Trees: Data-Oriented Translation. *Proceedings of the 9th Machine Translation Summit*, 165–172.

Daniel Jurafsky and James H. Martin. 2009. *Speech and Language Processing*. Pearson Prentice Hall, Upper Saddle River, NJ.

Philipp Koehn. 2005. Europarl: A Parallel Corpus for Statistical Machine Translation. *Proceedings of the 10th Machine Translation Summit*, 79–86.

Sotiris Kotsiantis and Dimitris Kanellopoulos. 2006. Association Rules Mining: A Recent Overview. *GESTS International Transactions on Computer Science and Engineering*, 32(1):71–82.

Fabrizio Luccio, Antonio Enriquez, Pablo Rieumont and Linda Pagli. 2001. Exact Rooted Subtree Matching in Sublinear Time. *Università Di Pisa Technical Report TR-01-14*.

Fabrizio Luccio, Antonio Enriquez, Pablo Rieumont and Linda Pagli. 2004. Bottom-up subtree isomorphism for unordered labeled trees. *Università Di Pisa Technical Report TR-04-13*.

Edward M. McCreight. 1976. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272.

Igor A. Mel'čuk. 1988. *Dependency Syntax: Theory and Practice*. State University of New York Press, Albany, NY.

Arjen Poutsma. 2000. Data-Oriented Translation. *Proc. of the 18th International Conference on Computational Linguistics (COLING 2000)*, 635–641.

Arjen Poutsma. 2003. Machine Translation with Tree-DOP. *Data-Oriented Parsing*, 63–81.

Deepak Ravichandran and Eduard Hovy 2002. Learning surface text patterns for a question answering system. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL-02)*, 41–47.

Kensei Sugayama and Richard Hudson, editors. 2006. *Word Grammar: New Perspectives on a Theory of Language Structure*. Continuum International Publishing, London.

Lucien Tesnière. 1959. *Eléments de la syntaxe structurale*. Editions Klincksieck, Paris.

John Tinsley, Mary Hearne and Andy Way. 2007. Exploiting Parallel Treebanks for use in Statistical Machine Translation. *Proceedings of Treebanks and Linguistic Theories (TLT '07), Bergen, Norway* 175–187.

Gertjan van Noord. 2006. At last parsing is now operational. *Verbum Ex Machina. Actes de la 13e conférence sur le traitement automatique des langues naturelles (TALN6)*, 20–42.

Peter Weiner. 1973 Linear pattern matching algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*, 1–11.

Mohammed J. Zaki. 2002. Efficiently mining frequent trees in a forest. *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1021–1035.

Ventsislav Zhechev and Andy Way. 2008. Automatic Generation of Parallel Treebanks. *Proceedings of the 22nd International Conference on Computational Linguistics (COLING 2008)*, 1105–1112.