Varro: An Algorithm and Toolkit for Regular Structure Discovery in Treebanks

Scott Martens
Centrum voor Computerlinguïstiek, KU Leuven
scott@ccl.kuleuven.be

Abstract

The *Varro* toolkit is a system for identifying and counting a major class of regularity in treebanks and annotated natural language data in the form of tree-structures: frequently recurring unordered subtrees. This software has been designed for use in linguistics to be maximally applicable to actually existing treebanks and other stores of tree-structurable natural language data. It minimizes memory use so that moderately large treebanks are tractable on commonly available computer hardware. This article introduces *condensed canonically ordered trees* as a data structure for efficiently discovering frequently recurring unordered subtrees.

1 Credits

This research is supported by the AMASS++ Project[1] directly funded by the *Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)* (SBO IWT 060051).

2 Introduction

Treebanks and similarly enhanced corpora are increasingly available for research, but these more complex structures are resistant to the techniques used in NLP for the statistical analysis of strings. This paper introduces a new treebank analysis suite *Varro*, named after Roman philologist Marcus Terentius Varro (116 BC-27 BC), who made linguistic regularity and irregularity central to his philosophy of language in *De Lingua Latina*. (Harris and Taylor, 1989)

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

The *Varro* toolkit focuses on a general problem in performing statistical analyses on treebanks: identifying, counting and extracting the distributions of frequently recurring unordered subtrees in treebanks. From this base, it is possible to construct more linguistically motivated schemes for performing treebank analysis. Complex statistical analyses are constructed from knowledge about frequency and distribution, so this constitutes a low level task on top of which higher level analyses can be performed.

An algorithm that can efficiently extract frequently recurring subtrees from treebanks has a number of immediate applications in computational linguistics:

- Speeding up treebank search algorithms like Tgrep2. (Rohde, 2001)
- Rule discovery for tree transducers used in parsing and machine translation. (Knight and Graehl, 2005; Knight, 2007)
- Generalizing lexical statistics techniques in NLP – e.g., collocation – to a broader array of linguistic structures. (Sinclair, 1991)
- Efficiently identifying useful features for tree kernel methods. (Moschitti, 2006)

3 Theory and Previous Work

For the purposes of this paper, a treebank is any collection of disjoint labeled trees. While in practice this mostly means parsed natural language sentences, the approach described here is equally applicable to other kinds of data, including semantic feature structures, morphological analyses, and doubtless many other kind of linguistically motivated structures. Figure 1 is an example of a parse tree from a Dutch-language treebank.

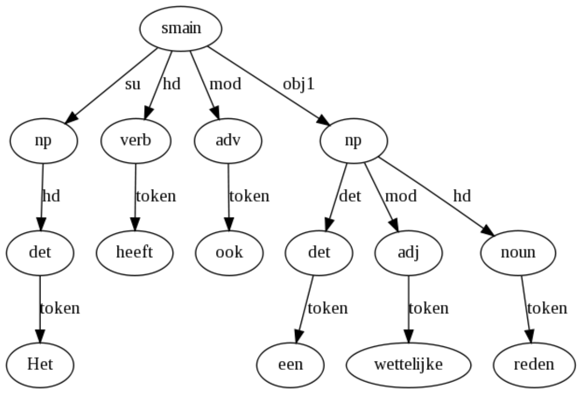
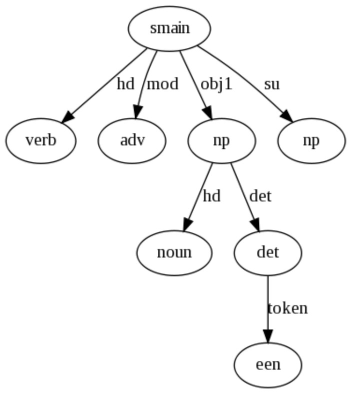


Figure 1: A tree from the Europarl Dutch corpus. (Koehn, 2005) It has been parsed and labeled automatically by the Alpino parser. (van Noord, 2006) A word-for-word translation is "*It also has a legal reason.*" (≈ "*There is also a legal reason (for that).*")

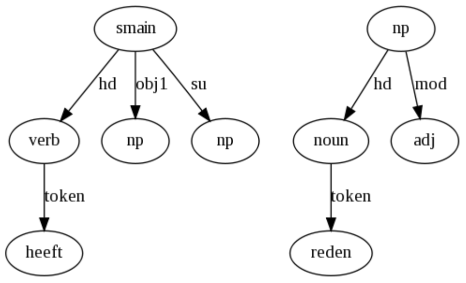
In this paper, we are concerned with identifying and counting *frequent induced unordered subtrees* in treebanks. The term *subtree* has a number of definitions, but this paper will follow the terminology of Chi et al. (2004). Figure 2 contains three examples of *induced unordered subtrees* of the tree in Figure 1. Note that the ordering of the vertices in the subtrees is different from that of Figure 1. This is what makes them *unordered subtrees*. *Induced subtrees* are more formally described in Section 4.

3.1 Apriori

The research builds on frequent subtree discovery algorithms based on the well-known *Apriori* algorithm, which is used to discover frequent itemsets in databases. (Agrawal et al., 1993) As a brief summary of *Apriori*, consider a collection of ordered itemsets $\mathbb{C} = \{\{a,b,c\},\{a,b,d\},\{b,c,d,e\}\}$. *Apriori* discovers all the subsets of those elements that appear at least some user-determined θ times. As an example, let us set $\theta = 2$, and then count the number of times each unique item appears in $\mathbb{C}$. Any single element in $\mathbb{C}$ that appears less than two times cannot be a member of a set of elements that appears at least θ times (since $\theta = 2$), so those are rejected. Each of the remaining set elements $\{a,b,c,d\}$ is extended by counting the number of two-element sets that include it and some element to the right in the ordered itemsets in $\mathbb{C}$. For b, these are $\{\{b,c\},\{b,d\},\{b,e\}\}$. Of this set, only those that appear at least θ times are retained: $\{\{b,c\},\{b,d\}\}$. This process is repeated for size three sets, and iterated over and over for increasingly large subsets, until there are no extensions that appear at least θ times. This whole procedure is then repeated for each unique item. Finally, *Apriori* will have extracted and counted all itemsets that appear at least θ times in $\mathbb{C}$.



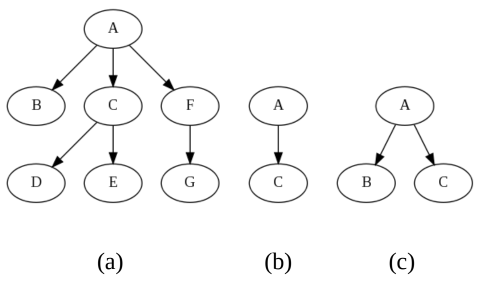
(a)



(b) (c)

Figure 2: Three induced unordered subtrees of the tree in Figure 1

Extending *Apriori* to frequent subtree discovery dates to the work of Zaki (2002) and Asai et al. (2002). Chi et al. (2004) summarizes much of this line of research. In *Apriori*, larger and less frequent itemsets are discovered



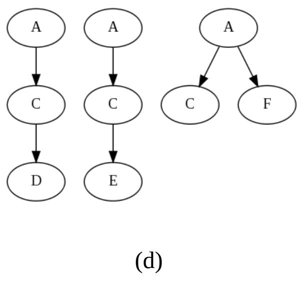


Figure 3: 3(b) and 3(c) are a subtrees of 3(a). The subtrees in 3(d) are possible extensions to 3(b), while 3(c) is not.

and counted by adding items to shorter and more frequent ones. This extends naturally to trees by initially locating and counting all the one-vertex trees in a treebank, and then constructing larger trees by adding vertices and edges to their right sides.

In Figure 3, subtree 3(b) has as valid extensions subtrees 3(d), all of which extend 3(b) to the right. An extension like subtree 3(c), which adds a node to the left of the rightmost node of 3(b), is not a valid extension.

3.2 Treebank applications

Applying these algorithms to natural language treebanks, however, presents a number of challenges.

The approach described above, because it constructs and tests subtrees by moving from left to right, is well-suited to finding *ordered subtrees*. However, this paper will consider *unordered subtrees* as better motivated linguistically. Word order is not completely fixed in any language, and can be very free in many important contexts.

But there are other problems as well. Apriori-style algorithms have the general property that their run-time is proportionate to the size of the output. Given a data-set D and a user-determined minimum frequency threshold θ, this class of solution outputs all the patterns that appear at least θ times in D. If D contains n patterns that appear at least θ times, $\mathbb{P} = \{p_1, p_2, ..., p_n\}; \forall p_i \in \mathbb{P} : freq(p_i) \geq \theta$, then the time necessary to identify and count all the patterns in $\mathbb{P}$ is proportionate to $\sum_{i=1}^{n} freq(p_i)$. In weakly correlated data, this is a very efficient method of finding patterns. In highly correlated data, however, the number of patterns present can become prohibitively large and extend run-time to unacceptable lengths, especially for small θ or large data-sets. Each frequent pattern may have any number of sub-patterns, each of which is also frequent and must be separately counted.

If we identify patterns with subtrees, a subtree with n vertices will, depending on its structure, have a minimum of $n(n-1)$ and a maximum of $(n-1)! + 1$ subtrees. If each of those subtrees is also a pattern that must be counted, then runtime grows very rapidly even for very small data-sets. Since natural language data is highly correlated, simple subtree-discovery extensions of *Apriori*, like those proposed in (Zaki, 2002) and (Asai, 2002), are not feasible for linguistic use. As reported in Martens (2009b), run-times become intractably long very quickly as data size increases for really existing treebanks.

However, there are compact representations of frequent patterns that are better suited to highly-correlated data and which can be efficiently discovered by modified *Apriori* schemes. This paper will only address one such representation: *frequent closures*. (Boulicaut and Bykowski, 2000) Frequent closures are widely used in subtree discovery and have an intuitive meaning when discussing natural language.

Given a treebank D, and a tree T that has a support of $freq(T) = \theta$, then T is *closed* if there is no supertree $T' \supset T$ where $freq(T') = \theta$. In Figure 3, if subtree 3(c) is as frequent in some treebank as 3(b), then 3(b) is not a closed subtree, nor can any further extension of it to the right be a closed subtree.

As a natural language example, given a corpus of English sentences, let us assume we have found a pattern of the form *"NP make up NP to VP"*, such as in *"He has made up his mind to study linguistics."* If every time this pattern appears in the corpus, the second NP contains *"mind"*, then the pattern is *not* closed. A larger pattern appears just as often and in exactly the same places.

This makes the notion of frequent closed subtree discovery a generalization of *collocation* and *coligation* - well known in corpus-based lexicography - to arbitrary tree structures. (Sinclair, 1991) J.R. Firth famously said, "You shall know a word by the company it keeps." (Firth, 1957) Frequent subtree discovery tells us exactly what company entire linguistic structures keep.

3.3 Efficient closed subtree discovery

Chi et al. (2005a) outlines a general method for efficiently finding frequent closed subtrees without finding all frequent subtrees first. Their approach requires each subtree found to be aligned with its supertree before checking for closure and extensions. However, the alignment between a subtree and its supertree - the map from subtree vertices to supertree vertices - is not necessarily unique. A subtree may have a number of possible alignments with its supertree, even if one or more of the vertex alignments is specificed, as shown in Figure 4, which uses an example from the hand-corrected Alpino Treebank of Dutch.[2]

This can only be avoided by adding a restriction to trees: the combination of edge and vertex labels for each child of a vertex must be unique. This guarantees that specifying just one vertex in the alignment of a subtree to its supertree is enough to determine the entire unique mapping, but it is incompatible with most linguistic theories. Processes like tree binarization can meet this requirement, but only with some loss of generality: Some frequent closed subtrees in a collection of trees like Figure 4(a) will no longer be frequent, or will be less frequent, in a collection of binary trees.

Martens (2009a) describes an alternative method of checking for closure which does not require alignment and can, consequently, be much faster. It has, however, two drawbacks: First, it does not find all frequent closed unordered subtrees. Figure 5 shows the kind of tree where that approach is unable to correctly identify and count an unordered subtree. Second, it requires a great deal more memory than solutions that align each subtree discovered and check directly for closure, and is therefore of limited use with very large corpora.

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

4 Definitions

A *fully-labeled rooted tree* is a rooted tree in which each vertex and each edge has a label: $T := \langle V, E, L_V, L_E \rangle$, where V is the set of vertices, E is the set of edges, L_V is a map $L_V : V \rightarrow \mathbb{L}_V$ from the vertices to a set of labels; and similarly L_E maps the edges to labels $L_E : E \rightarrow \mathbb{L}_E$. We will designate an edge e connecting vertex v_1 to its child v_2 by the notation $e = \langle v_1, v_2 \rangle$. $\mathbb{L}_V$ and $\mathbb{L}_E$ constitute collectively the *lexicon*. Figure 1 is an example of a fully-labelled, rooted tree from a Dutch-language treebank. This formalization is broadly applicable to all linguistic formalisms whose structures are *tree-based* or can be converted one-to-one into trees without loss of generality. This may require some degree of restructuring of the tree formats used in particular linguistic theories. For example, in many formal linguistic theories, labels are not atomic symbols, but may have many parts or even whole structured feature sets. In general, these can be mapped to trees with atomic labels by inserting additional vertices, or by taking advantage of edge labelling.

The algorithm described here is insufficient for formal structures that require more powerful graph formalisms like directed acyclic graphs.

The relations *parent*, *child* and *sibling* are taken here in their ordinary sense in discussing trees. In Figure 1, the vertex labeled *adv* is a *child* of the vertex labeled *smain*, the *parent* of the vertex labeled *ook*, and a sibling of the vertex labeled *verb* and the two vertices labeled *np*. To simplify definitions, the operator $label(x)$ will indicate the label of vertex or edge x.

An *induced unordered subtree* is a connected subset of the vertices of some tree that preserves the vertex and edge labels and the parent-child relations of that tree but need not preserve the ordering of siblings. Given a fully-labeled tree $T := \langle V_T, E_T, L_{V_T}, L_{E_T} \rangle$, an *induced subtree* S of T is

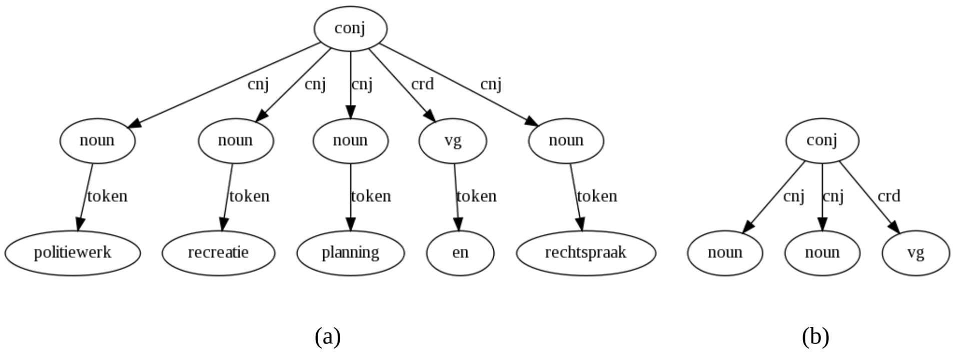


Figure 4: In 4(a) is a Dutch phrase conjoining multiple nouns. It translates as *"police work, recreation, planning and court activities"*. 4(b) has six unique unordered alignments with 4(a).

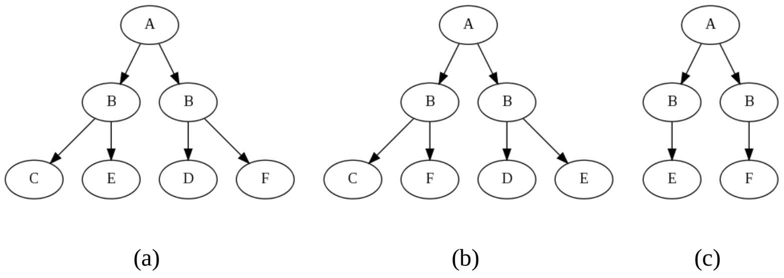


Figure 5: Subtree 5(c) is an unordered subtree of both 5(a) and 5(b), but the algorithm described in Martens (2009a) is unable to capture this in all cases.

a fully-labeled tree $S := \langle V_S, E_S, L_{V_S}, L_{E_S} \rangle$ for which there is an injection $M : V_S \rightarrow V_T$ from the vertices of S to some subset of the vertices of T, and for which:

$\forall v \in V_S$:

a. $label(v) = label(M(v))$
b. $e = \langle parent(v), v \rangle \in E_S \rightarrow$
   $e' = \langle M(parent(v)), M(v) \rangle \in E_T$
c. $label(e) = label(e')$

See Figures 1 and 2 for examples of subtrees of a particular tree.

We will further define all subtrees that are identical except in the ordering of their vertices to be *unordered isomorphic*. If a tree T is a subtree of tree T', we will follow set notation by denoting this relation as $T \subseteq T'$.

4.1 Canonical Ordering

Using canonical orderings to solve frequent unordered subtree problems was first proposed in Luccio et al. (2001) and expanded by other researchers in frequent subtree discovery techniques, notably in Chi et al. (2005b). Since the *Apriori*-style approaches described in Section 3.1 are suited only to finding subtrees whose vertices appear in a particular order, this paper will describe a mechanism for converting fully-labeled trees into canonical forms that guarantee that all instances of any unordered subtree will have an identical order to their vertices.

We must first define a strict total ordering over vertex and edge labels. Given lexica for the edge and vertex labels, $\mathbb{L}_E$ and $\mathbb{L}_V$ respectively, we define a strict total ordering on each such that $\forall l_i, l_j \in \mathbb{L}$ either $l_i \prec l_j$ or $l_i \succ l_j$ or $l_i = l_j$ and if $l_i \prec l_j$ and $l_j \prec l_k$, then $l_i \prec l_k$.

In a collection of fully-labeled trees, every vertex v that is not the root of some tree can be associated with a *full label* which is the pair $fullLabel(v) = \langle label(\langle parent(v), v \rangle), label(v) \rangle$, containing the label of the edge leading to its parent and the label of the vertex itself. For any pair of vertices where the edge to their parent is different, we order the vertices by the order of those edges. Where the edges are the same, we order them by the ordering of their vertex labels. Where we have two sibling vertices v_i and v_j such that $fullLabel(v_i) = fullLabel(v_j)$, we recursively order the descendants of v_i and v_j, and then compare them. In this way, two nodes can only have an undefined order if they have both exactly the same full labels and identical descendants.

A *canonically ordered tree* is a tree $T := \langle V_T, E_T, L_{V_T}, L_{E_T} \rangle$, where for each $v \in V_T$, the children of v are ordered in just that fashion.

4.2 Condensed trees

A *condensed tree* is a fully-labeled tree $T := \langle V_T, E_T, L_{V_T}, L_{E_T} \rangle$ with two additional properties:

a. Each vertex $v \in V$ is associated to a list of indices $parentIndex(v) = \{i_1, i_2, ..., i_n\}$, which we will call its *parent index*. Each entry $i_1, i_2, ..., i_n$ is a non-negative integer.
b. No vertex $v \in V$ has two children with the same full label.

Condensed trees are constructed from non-condensed trees as follows:

Given a tree $T := \langle V, E, L_V, L_E \rangle$, we first canonically order it, as described in the previous section. Then, we attach a parent index to each vertex $v \in V$ which is not the root of T. The initial parent index of each node consists of a single zero.

We then traverse the vertices of the now ordered tree T in breadth-first order from the the root downwards and from left to right. Given some $v_j \in V$, if it has no sibling to its right, or if the sibling to its immediate right has a different vertex label or a different edge label on the edge to its parent, we do nothing. Otherwise, if v_j has a sibling to its immediate right v_i with the same full label, we set ℓ_i to the size of $parentIndex(v_i)$, and then we append the $parentIndex(v_j)$ to $parentIndex(v_i)$. Then, we take the children of v_j, and for each one, we increment each value in its parent index by ℓ_i, and then insert it under v_i as one of v_i's children. We delete v_j and then we reorder the children of v_i into the canonical order defined in Section 4.1.

This is performed in breadth-first order over T. The result is guaranteed to be a tree where each vertex never has two children with the same edge and vertex labels. Figure 6 shows how the trees in Figure 5 look after they are converted into condensed trees. We will denote condensed trees as $\mathfrak{T} = cond(T)$, to indicate that $\mathfrak{T}$ has been constructed from T.

If two non-condensed trees are unordered isomorphic, then their condensed forms will be identical, including in their vertex orderings and parent indexes. If two condensed trees are identical, then the non-condensed trees from which they are constructed are always unordered isomorphic.

Each vertex $\mathfrak{v}$ of a condensed tree $\mathfrak{T} = cond(T)$ has a parent index containing some number of entries corresponding to a set of vertices in non-condensed tree T. We will designate that set as $orig(\mathfrak{v})$, a subset of the vertices in T. Given a condensed tree vertex $\mathfrak{v}$ and its parent $\mathfrak{p}$, if the size of $orig(\mathfrak{p})$ is larger than one, then the vertices in $\mathfrak{v}$ may have different parents in T. We can interpret the integers in the parent index of each condensed tree vertex as indicating which parent each member of $orig(\mathfrak{v})$ has.

In this way, given $\mathfrak{T} = cond(T)$, there is a one-to-one mapping from the vertices of T to a pair $\langle \mathfrak{v}, i \rangle$ consisting of some vertex in $\mathfrak{T}$ and an index to an entry in its parent index. If some vertex v in T maps to $\langle \mathfrak{v}, i \rangle$, then all the children of v, $c \in children(v)$ map to pairs $\langle \mathfrak{c}, j \rangle$ such that $parent(\mathfrak{c}) = \mathfrak{v}$ and the jth entry in $parentIndex(\mathfrak{c})$ is i. We can use this to define parent-child operations over condensed trees that perfectly match parent-child operations in non-condensed ones.

We will define a *skeleton tree* as a condensed tree stripped of its parent indices, and denote it as $skel(\mathfrak{T})$. Note that for any non-condensed tree T and any non-condensed subtree $S \subseteq T$, $skel(cond(T))$ will always contain $skel(cond(S))$ as an *ordered* subtree, including in cases like Figure 5, as shown in Figure 6.

4.3 Alignment

An alignment of a condensed subtree $\mathfrak{S}$ with a condensed tree $\mathfrak{T}$ has two parts:

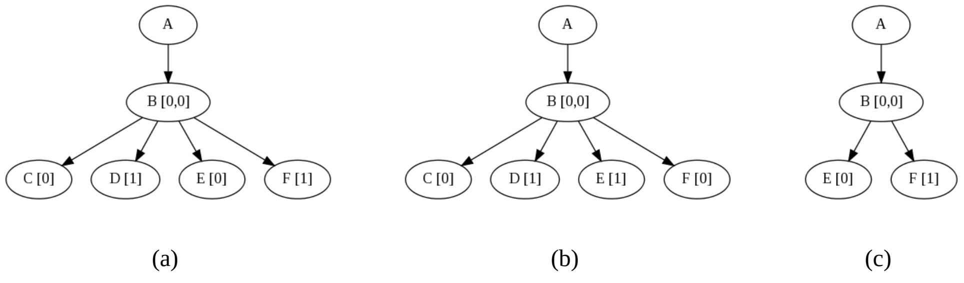


Figure 6: The trees in Figure 5 transformed into their condensed equivalents, with their parent arrays. Note that 6(c) is visibly an ordered subtree of both 6(a) and 6(b) if you ignore the parent arrays.

a. Skeleton Alignment:
   An injection $M : V_{\mathfrak{S}} \rightarrow V_{\mathfrak{T}}$ from the vertices of $\mathfrak{S}$ to the vertices of $\mathfrak{T}$.
b. Index Alignment:
   For each vertex $v_{\mathfrak{S}} \in V_{\mathfrak{S}}$, a bipartite mapping from the vertices in $orig(v_{\mathfrak{S}})$ to the vertices in $orig(M(v_{\mathfrak{S}}))$.

The first part is an alignment of $skel(\mathfrak{S})$ with $skel(\mathfrak{T})$. Given an alignment from the root of $\mathfrak{S}$ to some vertex in $\mathfrak{T}$, this can be performed in time proportionate, in the worst case, to the number of vertices in $skel(\mathfrak{T})$. If all the parent indices of the aligned vertices in the subtree and supertree have only one index in them, then the index alignment is trivial and the alignment of $\mathfrak{S}$ to $\mathfrak{T}$ is complete.

In other cases, index alignment is non-trivial. The method here draws on the procedure for unordered subtree alignment proposed by Kilpeläinen (1992). In the worst case, it resolves to the same algorithm, but can perform better on the average because of the structure of condensed trees.

Alignment proceeds from the bottom-up, starting with the leaves of $\mathfrak{S}$. If vertex $\mathfrak{s}$ is a leaf of $\mathfrak{S}$ and is aligned to some vertex $\mathfrak{t}$ in $\mathfrak{T}$, then we initially assume any member of $orig(\mathfrak{s})$ can map to any member of $orig(\mathfrak{t})$. We then proceed upwards in $\mathfrak{S}$, checking each vertex $\mathfrak{s}$ in $\mathfrak{S}$ to find a mapping from $orig(\mathfrak{s})$ to $orig(\mathfrak{t})$ such that if some $s \in orig(\mathfrak{s})$ can be mapped to some $t \in orig(\mathfrak{t})$, then the children of s can be mapped to children of t.

Once we reach the root of $\mathfrak{S}$, we proceed back downwards, removing those mappings from each $orig(\mathfrak{s})$ to its corresponding $orig(\mathfrak{t})$ that are impossible because their parents do not align.

The remaining index alignments must still be checked to verify that each one can form a part of a one-to-one mapping from $orig(\mathfrak{s})$ to $orig(\mathfrak{t})$. This is equivalent to finding a maximal bipartite matching from $orig(\mathfrak{s})$ to $orig(\mathfrak{t})$ for each possible alignment from $orig(\mathfrak{s})$ to $orig(\mathfrak{t})$. Bipartite matching is a problem with a number of well-documented solutions. (Dijkstra (1959), Lovász (1986), among others)

5 Algorithm

Having outlined condensed trees and how to align them, we can build an algorithm for extracting all frequent closed unordered subtrees from a treebank of condensed trees, given a minimum frequency threshold θ. Space restrictions preclude a full formal description of the algorithm, but it closely follows the general outline for closed tree discovery schemes advanced by Chi et al. (2005a):

1. Pass through the treebank collecting all the subtrees that consist of a single vertex label and all their locations.
2. Remove those that appear less than θ times.
3. Loop over each remaining subtree, aligning it to each place it appears in the treebank
4. Collect all the possible extensions, creating a new list of two vertex subtrees and all their locations.
5. Use the extensions to the left of the rightmost vertex in each alignment to check if the subtree is closed to the left, and reject it if it is not.
6. Use the extensions to the right of the rightmost vertex to check if the subtree is closed to the right, and output it if it is.

7. Retain the extensions to the right of the rightmost vertex and their locations if those extensions appear at least θ times.
8. Repeat for those subtrees.

6 Implementation and Performance

The *Varro* toolkit implements condensed trees and the algorithm described above in Python 3.1 and has been applied to treebanks as large as several hundred thousand sentences. The software and source code is available from sourceforge.net[3] and includes a small treebank of parsed Latin texts provided by the Perseus Digital Library. (Bamman and Crane, 2007)

The worst case memory performance of this algorithm is $O(nm)$ where n is the number of vertices in the treebank and m is the largest frequent subtree found in it. However, only the most pathologically structured treebank could come close to this ceiling, and in practice, the current implementation has so far never used as much twice the memory required to store the original treebank.

The runtime performance is, as described in Section 3.2, proportionate to the size of the output. However, aligning each occurrence of each subtree adds an additional factor. Given a condensed subtree $\mathfrak{S}$ and its condensed supertree $\mathfrak{T}$ containing $size(\mathfrak{T})$ vertices, and one already aligned vertex, the worst case alignment time is $O(size(\mathfrak{T})^{2.5})$, but only a highly pathological tree structure would approach this. The best case alignment time is $O(size(\mathfrak{S}))$. Therefore, it always takes more time to align larger subtrees, and since larger subtrees are less frequent than smaller ones, setting lower minimum frequency thresholds increases the average time required to process a subtree.

Processing even the small Alpino Treebank produces very large numbers of frequent closed subtrees. After removing punctuation and the tokens themselves, leaving just parts-of-speech and consituency labels - the Alpino treebank's 7137 sentences are reduced to 206,520 vertices. Within this small set, *Varro* took 1252 seconds to find 7307 frequent closed subtrees that appear at least 100 times. This is both considerably more subtrees than reported by Martens (2009b) on the same data and considerably more time.

[3]http://varro.sourceforge.net/

Speed and memory performance are the major practical issues in this line of research. Choosing to design *Varro* with memory footprint minimization in mind is a source of some performance bottlenecks. Using Python also takes a heavy toll on speed and a C++ implementation is planned. The fast alignment-free closure checking scheme in Martens (2009b) can also be implemented using condensed trees. On small treebanks this will improve speed without loss of precision, but has limited applicability to large treebanks.

7 Conclusions

The trade-off between memory usage, run-time and completeness for this kind of algorithm is *punitive*. The user must balance *very* long run-times against excessive memory usage if they want to accurately count all frequent unordered induced subtrees. The *Varro* toolkit is designed to make it possible to choose what tradeoffs to make. Since any subtree can be extended and checked for closure independently of other subtrees, *Varro* can easily implement heuristics designed to further reduce the number of subtrees extracted. We believe the future of this line of research lies in large part in that direction and hope that public release of *Varro* will aid in its development.

We have also discovered that there is a very strong relationship between the concision and consistency of linguistic formalisms and *Varro*'s performance. We restructured the Alpino data by promoting the head of each constituent, creating dependency-style trees along the lines described by Tesnière (1959) and Mel'čuk (1988). This reduced the number of subtrees found by 50%-60% and reduced run-times consistently by 60%-70% across a range of minimum frequency thresholds and treebank sizes. As a general rule, increasing the degree of linguistic abstraction increases the number of frequent subtrees, and consequently slows *Varro* down dramatically. Identifying linguistic formalisms that lend themselves to efficient and productive subtree discovery is another significant direction for this research, and one with immediate impact on other areas in linguistics.

References

Agrawal, Rakesh, 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*, pp. 207–216.

Asai, Tatsuya, 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.

Bamman, David and Gregory Crane. 2007. The Latin Dependency Treebank in a Cultural Heritage Digital Library. *Proceedings of the Workshop on Language Technology for Cultural Heritage Data*, LaTeCH 2007: pp. 33–40. http://nlp.perseus.tufts.edu/syntax/treebank/

Boulicaut, J.-F. and A. Bykowski. 2000. Frequent closures as a concise representation for binary data mining. *Knowledge discovery and data mining: current issues and new applications*, PAKDD 2000: pp. 62–73.

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

Chi, Yun, 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.

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

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Firth, J.R. 1957. *Papers in Linguistics*. London: OUP.

Harris, Roy and Talbot J. Taylor. 1989/1997. Varro on Linguistic Regularity. In *Harris and Taylor, Landmarks in Linguistic Thought I: The Western Tradition from Socrates to Saussure*. 2nd ed. London: Routledge. pp. 47-59.

Kilpeläinen, Pekka. 1992. *Tree Matching Problems with Applications to Structured Text Databases.* PhD dissertation. Univ. Helsinki, Dept. of Computer Science.

Knight, Kevin. 2007. Capturing practical natural language transformations. *Machine Translation*, 21:121–133.

Knight, Kevin and Graehl, Jonathan. 2005. An Overview of Probabilistic Tree Transducers for Natural Language Processing. *Proceedings of the 6th CICLing*, 1–24.

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

Lovász, László and M.D. Plummer. 1986. *Matching Theory*. Amsterdam: Elsevier Science.

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

Moschitti, Alessandro. Making tree kernels practical for natural language learning. *Proceedings of the 11th Conference of the European Association for Computational Linguistics* (EACL 2006), 113–120.

Mel'čuk, Igor A. 1988. *Dependency syntax: Theory and practice*. Albany, NY: SUNY Press.

Martens, Scott. 2009a. Frequent Structure Discovery in Treebanks. *Proceedings of the 19th Computational Linguistics in the Netherlands* (CLIN 19).

Martens, Scott 2009b. Quantitative analysis of treebanks using frequent subtree mining methods. *Proceedings of the 2009 Workshop on Graph-based Methods for Natural Language Processing (TextGraphs-4)*, 84–92.

Rohde, Douglas. 2001. *Tgrep2 User Manual*. http://tedlab.mit.edu/∼dr/Tgrep2

Sinclair, John. 1991. *Corpus, Concordance, Collocation*. Oxford: OUP.

Tesnière, Lucien. 1959. *Éléments de syntaxe structurale*. Paris: Éditions Klincksieck.

van Noord, Gertjan. 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.

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.