Frequent Structure Discovery in Treebanks

An efficient, practical, actually usable approach

Scott Martens

Centrum voor Computerlinguïstiek, KU Leuven[1]

Abstract

Discovering frequent structures within large natural language corpora is one of the core problems of corpus linguistics, but it is difficult to do for richly structured data. This paper describes a practical algorithm to extract frequent structures from treebanks or annotated corpora that can be represented as a tree structures. It extracts the most frequent structures first, so that not all structures have to be counted in order to find the most frequent ones. This algorithm assumes random constant-time access to all parts of the treebank and has space and time bounds broadly proportionate to the size of the output, which is not readily predictable in most cases. It is efficient enough to be usable with reasonable sized corpora using conventional desktop workstations.

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. However, corpora are increasingly annotated and annotation schemes grow more complex, encompassing diverse systems of features. Furthermore, there is growing use of *treebanks* - corpora which have been parsed either by automatic processes, manually or frequently by a combination of the two. A great deal of useful information is encoded in these more complex structured corpora, but access to it is very limited because n-gram and bag models are only applicable to sequences of discrete symbols - tokens, lemmas, part-of-speech tags, or other discrete categorical markers - but not to hierarchal structures.

It is trivially easy to count individual atomic elements in any body of symbolic data, and it is relatively straightforward to store the distribution information for individual elements or short fixed-length sequences. 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. However, as linguists, we already know that these methods are inadequate to fully model the information in texts.

Consider the problem of identifying *phrases* in aligned bilingual corpora and extracting their translations, like the English fixed phrase “to rain cats and dogs”,

[1]This research supported by the AMASS++ Project (http://www.cs.kuleuven.be/ liir/projects/amass/) IWT (SBO IWT 060051).

Proceedings of the 19th Meeting of Computational Linguistics in the Netherlands
Edited by: Barbara Plank, Erik Tjong Kim Sang and Tim Van de Cruys.
Copyright ©2009 by the individual authors.

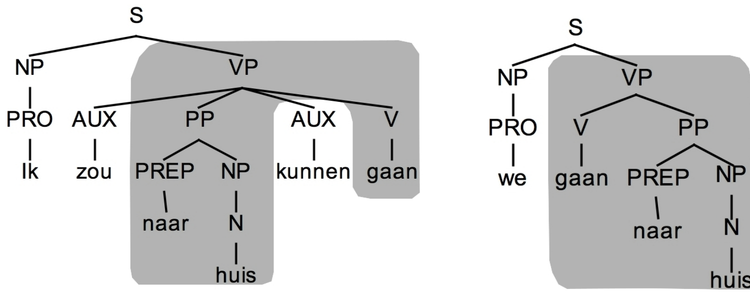


Figure 1: The two sentences “Ik zou naar huis kunnen gaan” and “We gaan naar huis”, parsed, with the repeated section highlighted. To save space, this tree does not take morphological variation into account and does not mark the difference between infinitive and and conjugated forms.

meaning *to rain very hard*. Many languages also have idioms that translate this phrase, none of which are even remotely related to a word for word translation. Translation discovery techniques that operate on *bag of words* assumptions are hopeless in this case. Suffix trees (or *tries*) and suffix arrays - not *suffix* in the linguistic sense but any tail of a sequence - can identify phrases like “rain cats and dogs” because they appear as *fixed phrases* with no variation in word order, little morphological variation, and no words inserted or removed. (Yamamoto and Church 2001)

However, most multiword structures are not fixed phrases. The English phrase “take ... into consideration” can appear with or without a freely varying element in the middle of it. A suffix array approach will not discover this phrase, even though it appears frequently, because it contains discontinuities. In a parsed treebank, all parts of this phrase are connected by dependency links and we should be able to identify this phrase as significant because it constitutes a repeated *connected subtree* in the treebank, much as a suffix array can identify “rain cats and dogs” because it is a repeated fixed phrase in a corpus.

A solution for finding repeated subtrees in treebanks also finds phrases with free word order, such as the Dutch usage “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 connected subtree that encompasses the phrase “naar huis gaan”, as in Figure 1.

Processes that treat texts as bags of words, or that use only very local sequential information like n-grams, will miss many important phenomena that are present as connected subtrees. This paper outlines an efficient and usable approach to

identifying frequently reoccurring connected subtrees in treebanks, but it is equally effective, given small modifications, to finding strings with gaps and to identifying other kinds of frequent correlations in richly structured data.

2 Closed structures

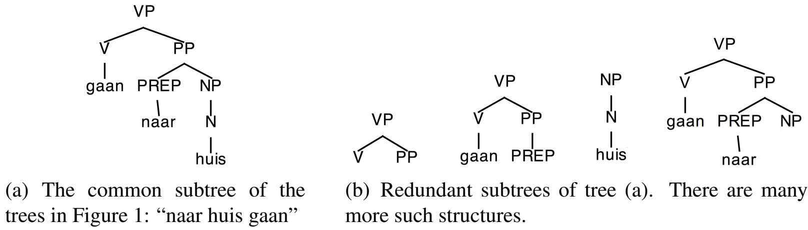
The problem of finding frequent structures like subtrees in tree structured data is hampered by the sheer number of possible structures contained even in very small datasets. The smaller tree in Figure 1 has 13 nodes, meaning it has at least $13^2 = 169$ and at most $12! = 479001600$ subtrees. A brute force approach to extracting the counts and distributions of all subtrees is not feasible even over very small treebanks.

Furthermore, many of the subtrees that a brute force approach could extract are *redundant*. 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 it is *redundant*, because it has 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.

For example, if we take the two sentences in Figure 1 as very small a treebank, there is only one closed subtree with a frequency of 2. However, that one closed subtree has a number of subtrees of its own that have the same frequency but are redundant, as shown in Figure 2.

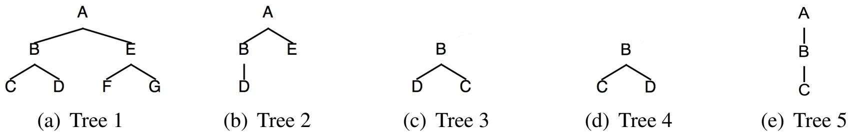
Closure is important for the discovery of translatable subsentential units. The phrase “rain cats and dogs” is translatable as a single unit, but “rain cats and” is not separately translatable, and will surely be incorrectly translated if treated in isolation. An algorithm which extracts only closed structures will never identify “rain cats and” as a potential unit, it will only identify “rain cats and dogs.” Similarly, for a structure like “naar huis gaan”, extracting only closed subtrees will ignore all the redundant subtrees that are part of it.



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

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

Figure 2: Closed and redundant subtrees.



(a) Tree 1 (b) Tree 2 (c) Tree 3 (d) Tree 4 (e) Tree 5

Figure 3: Trees 2, 3, 4 and 5 are a subtrees of Tree 1. Tree 3 is an *unordered* tree and Tree 4 is its ordered form. To construct a canonical representation, it is important to order the nodes, turning Tree 3 into Tree 4. Tree 5 is a tree that is automatically in ordered form.

3 Canonical string representations of trees

Much of the research in frequent subtree discovery has been based on *canonical string representations* of trees. The algorithm in this paper draws heavily on the *TreeMiner* and *FREQT* algorithms, both of which use canonical string representations. (Asai et al. 2002, Zaki 2002) The type of representation used here is called a *depth-first canonical form* (DFCF). In general, all depth-first canonical forms for trees are similar enough that the algorithms that use them can be readily translated to use a different one. The form used here is the one developed by Luccio et al. (2001).

The purpose of DFCF representations is to encode trees as strings in such a way that if one tree is a subtree of another tree, then there is an alignment of the DFCF of the smaller tree with that of the the larger. Using this type of representation, a number of tree matching problems are converted into string matching problems for which efficient solutions already exist. (See Luccio et al. (2001).)

To construct a DFCF for a tree, first sort the children of each node into a fixed lexicographic order. (See Figure 3.) This reordering is important because otherwise the DFCF of a tree will not necessarily be alignable with the DFCF of any subtree. The simplest way to explain the DFCF representation used here is to see it as equivalent to a LISP-style bracketed representation of a tree, but where only the end brackets of each constituent have to be indicated. In this paper, the end of each constituent is labeled with a 0. Trees 1 and 2 in Figure 3 are converted into DFCF as:

```
Tree 1:  ABC0D00EF0G000
Tree 2:  ABD00E00
```

Note that the number of zeros in each DFCF is equal to the number of node labels, and that a well formed tree must start with a node label and end with a zero.

Since Tree 2 is a subtree of Tree 1, there is an alignment of all the nodes and zeros of Tree 2 with nodes and zeros of Tree 1 such that the number of node labels and zeros in each unaligned part of Tree 1 are equal:

```
Tree 1:  ABC0D00EF0G000
Tree 2:  AB--D00E----00
```

This alignment can be performed in time proportionate to the size of the larger tree, the same as for string alignment with gaps.

DFCF representations are generally very compact and have the enormous advantage that they can be manipulated as strings. This form also makes it very easy to write fast functions for finding descendant, sibling and ancestor nodes of any node, and to extract whole sections of the tree.

The guarantee that a DFCF of a subtree will always be alignable with a larger tree that contains it only applies when the non-terminal children of each node have unique labels. This is not guaranteed to be true in natural language processing, but empirically appears to be usually true. To test this, we used the 7137 hand-corrected parse trees from the Alpino Treebank of Dutch.[1] Of the 230673 nodes in this sample, 3833 have more than one non-leaf child with the same label. See Figure 4 for an instance found in the Alpino treebank.

Nodes with non-uniquely labelled children were spread over 2666 of 7137 sentences. While this phenomenon affects a large minority of Alpino trees, in order to cause problems a frequently occurring subtree has to span at least two non-leaf children of a single node with the same label and at least one child of those nodes. Although difficult to empirically verify, this seems likely to be fairly a marginal phenomenon. Furthermore, for some applications this problem can be eliminated by using binary parse trees (cf. *Chomsky Normal Form* in Jurafsky and Martin (2009)). However, see Chi et al. (2005) for an algorithm that does not have this problem, but entails substantial extra processing costs. The simple, fast, memory-efficient tree generation scheme described below cannot be used when subtrees cannot be guaranteed to always appear with the same node order.

The principal value of depth-first canonical forms for this paper is that it is possible use them to quickly construct trees by generating and extending them to the right in DFCF representations. (Zaki 2002) Keeping count of the number of labels and 0s as we move across the DFCF representation, we can quickly identify which nodes can be attached to any subtree and where they are attached to.

Consider Tree 1 from Figure 3: $ABC0D00EF0G000$ and its subtree Tree 5: $ABC000$. (See Figure 3.) The rightmost node of Tree 5 is C, which aligns with

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

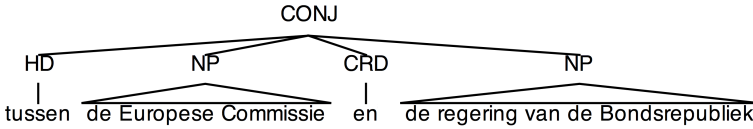


Figure 4: *"...between the European Commission and the government of the [German] Federal Republic..."* This structure is a simplified subtree of one of the sentences in the Alpino corpus where a node has two children with the same labels - two NPs. This can prevent the algorithm from discovering a small number of frequent subtrees.

address 2 in Tree 1:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Tree 1:	A	B	C	0	D	0	0	E	F	0	G	0	0	0
Tree 5:	A	B	C	0	–	–	–	–	–	–	–	–	0	0
alignment:			#											

To find all the nodes that can be added to the right of Tree 5, take the depth of the rightmost node in Tree 5 - which has a depth of 3 - and assign it to the address *following* the address of the node that aligns with the rightmost node of Tree 5. Then, if that address points to another node label, add 1 to the value at that address and assign the result to the *next* address. If that address points to a zero, then subtract 1 and assign it to the next address. Continue this way to the right until the end of Tree 1 is reached, as shown below:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Tree 1:	A	B	C	0	D	0	0	E	F	0	G	0	0	0
Tree 5:	A	B	C	0	–	–	–	–	–	–	–	–	0	0
depth:				3	2	3	2	1	2	3	2	3	2	1
maximum depth:				3	2	2	2	1	1	1	1	1	1	1
addable nodes:					#			#						

The addresses of the nodes that can be added to the right of Tree 5 to create a new connected subtree are those with values equal to the lowest value seen so far, when moving to the right, and that have labels rather than zeros in them. Furthermore, the value assigned to each address containing an addable node is the depth of its parent, so we know exactly where to attach it to Tree 5. Extending Tree 5 this way yields two new subtrees as possible extensions: $ABC0D000$ and $ABC00E00$, as shown in Figure 5.

If we have a subtree and we know the addresses of the rightmost node of each instance of it in a treebank, DFCF representations enable us to quickly identify extensions to that subtree without having to perform alignment or visit any ancestor node of the rightmost node in the subtree.

Figure 5: The two possible extensions of Tree 5 as a subtree of Tree 1. (Figure 3)

4 Algorithm and Data Structures

The algorithm described here is an *Apriori-style* approach (Agrawal et al. 1993) that builds heavily on the *TreeMiner* algorithm (Zaki 2002) for finding frequent subtrees. Unlike *TreeMiner*, this algorithm extracts *only* closed subtrees. It is not the first or only algorithm to do so - see Chi et al. (2004) for a more extensive summary of efforts to tackle this problem. The novelty of this approach is first, the manner in which it checks for closure, and second, its application to natural language treebanks.

4.1 Definitions

A treebank T consists of $|T|$ nodes belonging to a number of individual trees in DFCF representation. Each node has a label drawn from a lexicon L of size $|L|$. Each label in the treebank has a unique address a so that if the first label A of some subtree $AB0C00$ has an address a_0, then the label at address a_0+1 is B, the label at address a_0+2 is 0, etc. All addresses are sortable so that $a_0 < a_0+1 < a_0+2$. It is also assumed that the contents of the treebank are randomly accessible in constant, negligible time.

Each appearance of each subtree 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. or if those are the same that:
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.

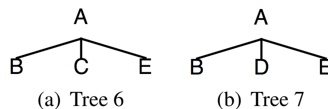
A *self-sorting queue* is any data structure that stores key-data pairs and stores the keys in order from greatest to least. The data structure used to implement a self-sorting queue in this research is an *AVL tree*, however, other structures could equally well have been used. B-trees in particular might well lead to improved performance. (See Knuth (1997) for broader discussion of this type of data structure.) 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 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, and where for each *Hit* in each *HitList*, the address of the root is the same as the address of the rightmost node. This set is constructed in linear time by iterating over all the nodes in the treebank.

It is important to note that the initial *HitLists* are constructed so that the *Hits* in them are in order from the first occurrence of the associated label to the last. This is important because the method of construction of larger subtrees ensures that each *HitList* built by extending an existing *HitList* will also have an ordered address list. It also means that the first appearance of any subtree in the treebank is always the first *Hit* in the associated *HitList*. 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.

To see this in action, take Trees 6 and 7 from Figure 6 to be a very small



(a) Tree 6 (b) Tree 7

Figure 6: A trivial two-tree treebank.

Table 1: (a) All *HitLists* in the example and (b) the state of the self-sorting queue after initialization. *Hits* are in *[root address, rightmost node address]* form.

(a) All initial *HitLists*

DFCF Tree	Hits
$A0$	[0,0], [8,8]
$B0$	[1,1], [9,9]
$C0$	[3,3]
$D0$	[11,11]
$E0$	[7,7], [15, 15]

(b) Initial queue state

DFCF Tree	queueKey
$A0$	[2, 15, 15, 1]
$B0$	[2, 14, 14, 1]
$E0$	[2, 10, 10, 1]

treebank. In DFCF form, with the addresses of each symbol noted, they are:

```
         0  1  2  3  4  5  6  7
Tree 6:  A  B  0  C  0  E  0  0

         8  9 10 11 12 13 14 15
Tree 7:  A  B  0  D  0  E  0  0
```

HitLists for each label, constructed as they would be when initialized, are shown in Table 1a. If we extract subtrees with a minimum frequency of 2, then we reject the *HitLists* for C and D since they only appear once. The remaining three *HitLists* are inserted into the queue, which sorts them in order of their *queueKeys*, as in Table 1b.

4.3 Extracting subtrees without checking for closure

Extracting all the subtrees above a fixed frequency, in order from the most frequent to the least, proceeds as follows:

Initialize as described above. Pop the top *HitList* from the queue and visit each *Hit*, getting all the valid right extensions and their addresses, and then constructing the set of *HitLists* for each new subtree. For each of the resulting new subtrees and their *HitLists*, check each to see if it has at least the specified minimum number of *Hits* in it. If so, insert it into the queue. If not, throw it away. This is essentially identical to the *TreeMiner* and *FREQT* algorithms already published by Zaki (2002) and Asai et al. (2002), except that it outputs frequent subtrees in order from the most frequent to the least.

To see Algorithm 1 *extractAllFrequentSubtrees* work, start with the state of the queue in Table 1. Remove the top entry in the queue, $A0$, and visit each instance of it, generating new subtrees as extensions to it by the means described in section 3. The result is four new subtrees, as shown in Table 2. Of these, only two have frequencies greater than or equal to the minimum frequency of 2, and are thus added to the queue. We also output $A0$ and its *HitList*.

Repeating this procedure until the queue is empty will output all subtrees that appear at least twice in this small treebank: $A0$, $AB00$, $AB0E00$, $AE00$, $B0$,

Algorithm 1
define **extractAllFrequentSubtrees**(minimum frequency t, Treebank T)

```
initialize empty self-sorting queue Q
fill Q with all single node subtrees with frequencies >= t
while length(Q) > 0 do
  currentHitList = pop top HitList from queue
  successors = rightExtendSubtree(hl)
  for all newHitList in successors do
    if countHits(newHitList) >= t then
      insert newHitList into Q
    end if
  end for
  output currentHitList
end while
```

Table 2: (a) Extensions to $A0$ and (b) the state of the queue after extending $A0$ and adding $A0$'s extensions to it.

(a) Extensions to $A0$

DFCF Tree	Hits
$AB00$	[0,1], [8,9]
$AC00$	[0,3]
$AD00$	[8,11]
$AE00$	[0,5], [8,13]

(b) Queue state

DFCF Tree	queueKey
$AB00$	[2, 15, 14, 2]
$AE00$	[2, 15, 10, 2]
$B0$	[2, 14, 14, 1]
$E0$	[2, 10, 10, 1]

$BE00$ and then $E0$.

4.4 Extracting only closed 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 a redundant subtree of some closed subtree with a frequency of f. If subtree x is a redundant subtree of tree y, both having the same frequency f, then there is some prefix of y, y_{prefix}, also with frequency f, for which the addresses of the rightmost nodes of every instance of y_{prefix} are identical to the addresses of the rightmost nodes of every instance of x. Furthermore, subtree x will be a subtree of y_{prefix}. This means that if we already know that y_{prefix} is a prefix of a closed subtree, we can check if x is a redundant subtree, and therefore

never store or extend it, by verifying that:

- x and y_{prefix} appear the same number of times in the treebank, and either:
- x is a subtree of y_{prefix}, or that:
- The set of addresses of the rightmost nodes of instances of x is *identical* to the set of addresses of rightmost nodes of instances of y_{prefix}.

The sort order of the self-sorting queue ensures that if a prefix of a closed subtree y_{prefix} is in the queue and some subtree of it x is also in the queue, then y_{prefix} is closer to the top of the queue than x is. For all redundant subtrees x and their corresponding prefix of a closed subtree y_{prefix}, $queueKey(y_{prefix})$ is greater than $queueKey(x)$. To ensure that y_{prefix} reaches the top of the queue before x, it suffices to guarantee that y_{prefix} is always added to the queue before x reaches the top.

The subtree y_{prefix} is generated by extending y_{prefix}'s immediate prefix y'_{prefix} - the subtree that is the same as y_{prefix} without its rightmost node. Subtree y'_{prefix} either appears exactly as often as y_{prefix} or it appears more often and the address of the root of the first instance of y'_{prefix} either precedes the address of the root of the first instance of y'_{prefix} or is identical to it, and the address of the rightmost node of the first instance of y'_{prefix} *must* precede that of y_{prefix} because y'_{prefix} is identical to y_{prefix} except that it has one less node on the right. In either case, y'_{prefix} precedes x in the queue if both are in the queue. This means that y'_{prefix} reaches the top of the queue before x, and y_{prefix} is inserted into the queue before x reaches the top. This logic extends back to all prefixes of y_{prefix} until we get to the single node subtree that corresponds to its root. All prefixes of y_{prefix} precede x in the queue, are extended before x reaches the top of the queue, and all their extensions that are at least as frequent as x reach the top of the queue before x does.

Ergo, every subtree that is not a closed subtree or a prefix of a closed subtree reaches the top of the queue *after* the closed subtree or prefix of a closed subtree that contains it and appears exactly the same number of times in exactly the same places. In order to prevent a redundant subtree from being stored or extended, it suffices to check each subtree as it reaches the top of the queue against the subtrees that have already reached the top of the queue. Algorithm 2 *extractAllFrequentClosedSubtrees* does just that.

Continuing the example from section 4.3 and applying Algorithm 2 *extractAllFrequentClosedSubtrees*, start with the initialized queue as in Table 1. $A0$ is the first subtree removed from the queue, and is a prefix of a closed subtree by virtue of being at the top of the queue when initialized. We get its extensions and insert them into the queue. (See Table 2.) However, we note that $A0$ appears twice, and its extensions $AB00$ and $AE00$ also appear twice. It is not, therefore, a closed subtree, so it is not outputted. However, its frequency and the list of addresses of its rightmost nodes - $[2, 0, 8]$ - *is* saved.

The next subtree removed from the top of the queue is $AB00$. It appears twice, with rightmost nodes at 1 and 9, so we check for $[2, 1, 9]$ in the table of subtrees

Algorithm 2

define **extractAllFrequentClosedSubtrees**(minimum frequency t, Treebank T)

```
initialize empty HitList repository R
initialize empty self-sorting queue Q
fill Q with all single node subtrees with frequencies >= t
while length(Q) > 0 do
  currentHitList = pop top HitList from queue
  if currentHitList does not match any HitList in R then
    store currentHitList in R
    successors = rightExtendSubtree(hl)
    for all newHitList in successors do
      if countHits(newHitList) >= t then
        insert newHitList into Q
      end if
    end for
    if no newHitList in successors is as frequent as currentHitList then
      output currentHitList
    end if
  else
    reject currentHitList and do nothing
  end if
end while
```

Table 3: Continuing from Table 2 after processing $AB0E00$ with *extractAllFrequent-ClosedSubtrees*: (a) the state of the queue, (b) the repository of information about previous subtrees, and (c) the subtrees already outputted.

(a) Queue state

DFCF Tree	queueKey
$AE00$	[2, 15, 10, 2]
$B0$	[2, 14, 14, 1]
$E0$	[2, 10, 10, 1]

(b) Stored information

Freq & addrs	Subtree
[2, 0, 8]	$A0$
[2, 1, 9]	$AB00$
[2, 5, 13]	$AB0E00$

(c) Output

DFCF Tree	Hits
$AB0E00$	[0, 5], [8, 13]

that have already reached the top of the queue. It does not match any - only $[2, 0, 8]$ is there - so it is extended.

$AB00$ has only one extension that appears at least twice: $AB0E00$, appearing with the *Hits* $[0, 5], [8, 13]$. Since it appears twice, $AB00$ is also not closed but it is a prefix of a closed tree, so $[2, 1, 9]$ is stored and nothing is outputted. $AB0E00$ is added to the queue. Since $AB0E00$ has a *queueKey* of $[2, 15, 10, 3]$, it goes to the top of the queue, ahead of $AE00$.

$AB0E00$ has no extensions and it does not match any previously processed subtree, so it *is* a closed subtree. We output it, and since it has no extensions, nothing is added to the queue, but we store its count and rightmost node addresses: $[2, 5, 13]$. Table 3 shows the state of the queue, the table containing information about previously processed subtrees, and the output up to this point.

So far we have only needed to check for closure by testing if any subtree has an

extension that occurs just as often, but with $AE00$ this will change. Its frequency is 2 and the addresses of its rightmost nodes are 5 and 13, so $AE00$ matches $AB0E00$. We therefore reject $AE00$ without extending it, without storing it anywhere, and without outputting it. A cursory look at the treebank (Figure 6) verifies that $AE00$ is not a closed subtree nor a prefix of a closed subtree. Every time $AE00$ appears, it is part of $AB0E00$.

In the same way, we reject the next subtree at the top of the queue, $B0$, because it matches $AB00$, and then we reject $E0$ for matching $AB0E00$. This empties the queue and the program stops, having outputted *only* the one closed subtree that appears twice in the corpus.

With closure checking, the runtime for this algorithm is proportionate to the total number of *Hits* in the all the *HitLists* inserted into the queue, which is somewhat more than all the *HitLists* outputted. Insertions to the queue include all closed subtrees, all prefixes of closed subtrees and all single-node extensions of closed subtrees and prefixes of closed subtrees. Memory usage includes all the *HitLists* in the queue at any one time, but also the hash table used to check for closure. This hash table can grow quite large, but can never be larger than the output.

It is not necessary to store all the rightmost node addresses of every *HitList* in order to verify closure, as in the example above. Only a subset is necessary and in practice we used only one address: the location of the rightmost node of the *first* appearance of each subtree. Other distribution-sensitive hashing schemes are possible and may improve performance. However, if all addresses are not stored, then more than one previously processed *HitList* may match a new *HitList*. To check for closure, the algorithm must then verify that a possibly redundant subtree is contained in a previously processed one. This is most effective at high frequencies, when address lists are long but trees are small and can be tested for containment quickly. At low frequencies - any frequency where the average number of nodes in a subtree is greater than the number of times that subtree appears - performance improves using complete address lists. Future work includes devising a more flexible distribution-sensitive hashing scheme for these conditions.

5 Results

Algorithm 2 *extractAllFrequentClosedSubtrees* was implemented using a mixture of Ruby[1] - an interpreted scripting language - and some C code. It was applied to the hand-corrected 7137 sentence subset of the Alpino Treebank of Dutch. The average sentence length in this small treebank is roughly 20 words, and the corresponding trees have an average of approximately 32 nodes.

Applying *extractAllFrequentClosedSubtrees* to this treebank, with the minimum frequency set to 2, yields 342,401 closed subtrees in about 2000 seconds of runtime on a conventional workstation running Linux. In total, 3,154,232 addresses were extracted - an average of just over 9 addresses per subtree. The frequent trees extracted contain 2,510,439 nodes - ten times as many as the total

[1] http://www.ruby-lang.org/

Table 4: Processing results using *extractAllFrequentClosedSubtrees*, *extractAllFrequentSubtrees*, and estimates for a naïve brute force solution.

Algorithm	**# of Subtrees**	**# of addresses**	**Total nodes in all subtrees**	**Runtime**
extractAllFrequentClosedSubtrees	342,401	3,154,232	2,510,439	2000 sec
extractAllFrequentSubtrees	4.2 million	14.1 million	124 million	11,000 sec
Naïve brute force	$< 6.3 \cdot 10^{34}$	$>$ one per subtree	$>$ # of subtrees	-

number of nodes in the treebank. At its peak, the queue contained roughly half as much data as the final output.

Algorithm 1 *extractAllFrequentSubtrees*, which does not check for closure, was also implemented and run on the same data with a minimum frequency threshold of 2. Table 4 compares it with *extractAllFrequentClosedSubtrees* and an estimate for a brute force approach. Eliminating checking for closure reduced the amount of time it took to process each address in the output, but increased the size of the output by much more. Tests of the same algorithms using subsets of the same treebank show that the difference between extracting all subtrees and extracting only closed subtrees grows very rapidly as treebank size grows. This follows logically because larger closed subtrees have more non-closed subtrees, and the larger a treebank becomes, the more large, low frequency closed subtrees will be discovered.

The hash table used to check for closure grew quite large towards the end of processing, since there are always more subtrees with low frequencies than with high frequencies, and low frequency subtrees will have more nodes than higher frequency ones. At low frequency thresholds, this tended to dominate memory usage as the program progressed.

Setting higher minimum thresholds reduces the size of the output and speeds up processing dramatically. Running *extractAllFrequentClosedSubtrees* with a threshold of 4 reduced the runtime to less than half what it was with a threshold of 2, roughly halved the number of subtrees found, and reduced the number of addresses outputted by a third. Peak memory usage was reduced by about two-thirds. This suggests that choosing the minimum frequency has a very substantial but non-linear effect on memory usage as well as run time. In all cases, the extra processing and memory required per subtree to find only closed subtrees dominates processing time at low frequencies.

6 Conclusions and future work

Porting this algorithm to a compiled language and optimizing the code is the first priority for further work in this field. Some of the results described here cannot be easily tested for their robustness until a faster implementation is in place. A previously implemented version of a very similar algorithm was used to extract frequent combinations of words in sentences from a database of news articles, rather than finding frequent subtrees, and rewriting a few frequently invoked functions in C

instead of Ruby improved performance roughly 20-fold. We expect comparable results for trees.

Using conventional desktop computers, this makes treebanks of hundreds of thousands of sentences accessible to comprehensive extraction of *all* reoccurring closed subtrees, including those appearing infrequently, and makes it feasible to process much larger treebanks with higher minimum frequency thresholds. This algorithm is also highly parallelizable, and we are investigating the possibility of running it over distributed hardware.

The non-linear effect of treebank size and minimum frequency thresholds on memory usage and runtime is one area that demands deeper investigation. Also, the empirical performance results described in the previous section may not transport well to treebanks using different linguistc formalisms than Alpino, and may vary depending on the language. Additionally, work is in progress to find an efficient solution for trees with non-unique child nodes.

The goals of this line of reseach include applications to natural language processing, notably to *Data Oriented Parsing* (Bod et al. 2003), which makes heavy use of subtree statistics. Applications to machine translation include using bilingual treebanks to extract subtree to subtree translations for the development of transfer rules. Statistical parsing can also be enhanced with access to subtree statistics by selecting parses that maximize the probabilities of their subtrees.

To date, there has been little corpus research using comprehensive censuses of frequently occurring structures instead of merely words. This research makes it possible to directly compare the frequencies of structures without having to specify those structures in advance. If it is a structure that takes the form of a reoccurring subtree in a treebank, it is accessible to this algorithm. Although a variety of statistical tools exist in corpus and computational linguistics for handling pairs of words and short sequences such as n-grams, many of these tools do not transport well to arbitrary structures. The development of new tools better suited to treebanks, particularly from within *Minimum Description Length* theory (Grünwald 2007), is also in progress as part of this research.

References

Agrawal, R., T. Imielinski, and A. N. Swami (1993), Mining association rules between sets of items in large databases, *Proc. 1993 ACM SIGMOD Intl. Conf. on Management of Data*, ACM Press, pp. 207–216.

Asai, T., K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa (2002), Efficient substructure discovery from large semi-structured data, *Proc. 2nd SIAM Intl. Conf. Data Mining (SDM '02)*, pp. 158–174.

Bod, R., K. Sima'an, and R. Scha, editors (2003), *Data-Oriented Parsing*, CSLI.

Chi, Y., R. Muntz, S. Nijssen, and J. Kok (2004), Frequent subtree mining - an overview, *Fundamenta Informaticae* **66** (1-2), pp. 161–198, IOS Press.

Chi, Y., Y. Xia, Y. Yang, and R. Muntz (2005), Mining closed and maximal frequent subtrees from databases of labeled rooted trees, *IEEE Transactions on Knowledge and Data Engineering* **17** (2), pp. 190–202, IEEE.

Grünwald, P. (2007), *The Minimum Description Length Principle*, MIT Press.

Jurafsky, D. and J. Martin (2009), *Speech and Language Processing*, Prentice Hall.

Knuth, D. (1997), *Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley Professional.

Luccio, F., A. Enriquez, P. Rieumont, and L. Pagli (2001), Exact rooted subtree matching in sublinear time, *Technical Report TR-01-14*, Università Di Pisa.

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

Zaki, M. J. (2002), Efficiently mining frequent trees in a forest, *Proc. Eighth Int'l Conf. Knowledge Discovery and Data Mining (ACM SIGKDD '03)*, ACM.