Fragment extraction with fast tree kernels

Andreas van Cranenburgh
acranenb@science.uva.nl

April 20, 2012

Abstract

This report details the implementation of a fragment extraction algorithm using an average case linear time tree kernel. Given a treebank, the algorithm extracts all fragments that occur at least twice, along with their frequency. Evaluation shows a 70-fold speedup over a quadratic fragment extraction implementation.

1 Introduction

A fragment extraction algorithm was first presented by Sangati et al. (2010). His algorithm compares each node in the input to all others, giving a best and worst-case quadratic complexity (i.e., $\Theta(n^2)$). A fast tree kernel was presented by Moschitti (2006), with an average case linear complexity (i.e., average $O(n)$). However, his version only returns a list of matching nodes. This work presents an implementation of the fast tree kernel that extracts recurring fragments, providing a significant speedup.

As an example, given these trees as input:

```
(S (NP (DT The) (NN cat)) (VP (VBP saw) (NP (DT the)
    (NP|<JJ-NN> (JJ hungry) (NN dog)))))
(S (NP (DT The) (NN cat)) (VP (VBP saw) (NP (DT the) (NN dog))))
```

We want the algorithm to find the following maximal common fragments:

```
(S (NP (DT The) (NN cat)) (VP (VBP saw) (NP ))) 2
(NP (DT ) (NN ))        2
(DT the)        2
(NN dog)        2
```

2 Iterating over the treebank

Each tree is represented as a list of nodes sorted according to its productions. For each pair of trees, a bit matrix is constructed where the bit at (n, m) is set iff the nodes at those indices in the respective trees have the same production. From this table the bitsets corresponding to fragments are collected and stored in the results table. We generalize the task of finding recurring fragments in a treebank to the task of finding the common fragments of two, possibly equal,

treebanks. In case the treebanks are equal, only half of the possible tree pairs have to be considered: the fragments extracted from $\langle t_n, t_m \rangle$, with $n < m$, are equal to those of $\langle t_m, t_n \rangle$.

The code in this report is in a superset of the Python language that includes type declarations; this allows translation of the code to C using Cython (Behnel et al., 2011).

```
def extractfragments(trees1, trees2):
    results = {}
    SLOTS = MAXNODE / (sizeof(ULong) * 8) + 1
    CST = bitmatrix(MAXNODE * MAXNODE) #Common Subtree Table
    for a in trees1:
        for b in trees2:
            # initialize table
            memset(CST, 0, b.len * SLOTS * sizeof(ULong))
            # fill table
            fasttreekernel(a.nodes, b.nodes, CST)
            # extract results
           getfragments(CST, a.nodes, b.nodes, b.root, results, SLOTS)
    return results
```

Some implementation details: we use arrays of `unsigned long` (abbreviated `ULong`) to store bit-vectors and bit-matrices. The trees are represented as arrays of `Node`s, which contain indices for their left and right descendants.[1] The index -1 is used as a sentinel value to indicate that there is no production or left node (terminals), or no right node (unary productions). This does mean that the representation requires binary trees, but this simplifies the algorithms and most statistical parsers rely on binary trees anyway. Since the bit-vectors and -matrices are dynamically allocated, we pass around pointers and perform manual indexing.

3 The fast tree kernel

The insight that makes this kernel fast on average is that it can be viewed as the problem of finding the intersection of two sorted sequences. In our implementation the productions of each node is mapped to an integer, so that comparisons are cheap. We sort in descending order, so that lexical nodes which have a sentinel production of -1 end up in the tail.

```
cdef void fasttreekernel(Node *a, Node *b, ULong *CST, int SLOTS):
    cdef int i = 0, j = 0, jj = 0
    while a[i].prod != -1 and b[j].prod != -1:
        if   a[i].prod < b[j].prod: j += 1
        elif a[i].prod > b[j].prod: i += 1
        else:
            while a[i].prod == b[j].prod:
                jj = j
                while a[i].prod == b[jj].prod:
```

1 Instead of array indices direct pointers could have been used, but these take four times as much memory: 8 bytes on a 64 bit machine, versus 2 bytes for a `short int`.

```
                    SETBIT(&CST[jj * SLOTS], i)
                    jj += 1
                i += 1
```

	0 VP	1 VBP	2 S	3 NP	4 NP	5 NN	6 NN	7 DT	8 DT
0 VP	1								
1 VBP		1							
2 S			1						
3 NP|<JJ-NN>									
4 NP									
5 NP				1	1				
6 NN						1			
7 NN							1		
8 JJ									
9 DT								1	
10 DT									1

Table 1: Matrix of two compared trees.

Given the trees from the introduction as input, the matrix in table 1 obtains. The matrix visualizes why the algorithm is efficient: there is a path along which comparisons have to be made, but most node pairs do not have to be considered. The more productions, the higher the efficiency. In case there is only a single non-terminal label X, and hence only one phrasal, binary production, the effciency of the algorithm disappears and the worst-case quadratic complexity will result.

4 Extracting maximal connected subsets

After the matrix with matching nodes has been filled, we need to extract nodes belonging to each maximal fragment. A fragment is a connected subset of nodes. We traverse the second tree in depth-first order, in search for possible root nodes of fragments.

To scan the bits of a row of the matrix we use the function `nextset`, which returns the next 1-bit starting from a specific index. This function exploits a CPU instruction which scans for the next 1-bit in a (64 bit) word sized chunk. While the extraction of bit sets technically has quadratic time complexity because we walk through a 2-dimensional matrix looking for 1-bits, in practice the bit operations are $O(1)$, not linear.

```
cdef void getfragments(ULong *CST, ULong *scratch, Node *a, Node *b,
    short j, dict results, int SLOTS):
    cdef short i
    if j < 0 or b[j].prod < 0: return
    while True:
        i = nextset(&CST[j * SLOTS], 0, SLOTS)
        if i == -1: break
        memset(scratch, 0, SLOTS * sizeof(ULong))
```

```
        extractat(CST, scratch, a, b, i, j, SLOTS)
        results[getfragment(tree, scratch)] += 1
    getfragments(CST, scratch, a, b, b[j].left, results, SLOTS)
    getfragments(CST, scratch, a, b, b[j].right, results, SLOTS)
```

Whenever a 1-bit is encountered, both trees are traversed in parallel from that node onwards, to collect the nodes. Both trees need to be considered to ensure that extracted subsets are connected in both trees.

```
cdef void extractat(ULong *CST, ULong *result, Node *a, Node *b,
    short i, short j, int SLOTS):
    SETBIT(result, i)
    CLEARBIT(&CST[j * SLOTS], i)
    if a[i].left < 0: return
    if TESTBIT(&CST[b[j].left * SLOTS], a[i].left):
        extractat(CST, result, a, b, a[i].left, b[j].left, SLOTS)
    if a[i].right < 0: return
    if TESTBIT(&CST[b[j].right * SLOTS], a[i].right):
        extractat(CST, result, a, b, a[i].right, b[j].right, SLOTS)
```

Here is an example which demonstrates why fragments must be extracted from both trees in parallel:

```
(S (A (B x) (S y)) (B p))
(A (B x) (S (A y) (B z)))
```

These trees have two productions in common; however, these do not form a contiguous fragment in both trees, because the productions appear in a different order in the respective trees.

Another concern is whether extracting fragments from a pair of trees is a commutative operation; i.e., whether the order of the operands has an effect on the output. Consider the following corpus:

```
(TOP (S (A x)) (S (A b)))
(TOP (S (A x)))
```

Using this order, we get two fragments with the `FragmentSeeker` of Sangati et al. (2010):

```
(S (A "x"))     1
(S A)   1
```

But when the order of the input is reversed, the second fragment is not extracted, because it is a subset of the first. With our algorithm, the output is the same regardless of the order of the input; two fragments are extracted from this example.

5 Evaluation

The work performed by the algorithm can be efficiently distributed among the available cores. To abstract over the number of available cores, we not only

report the wall clock time but also the total CPU time used by all cores, which is comparable to the time that would have been spent if a single core was available. As treebank we use the Wall Street journal section of the Penn treebank (Marcus et al., 1993). In a pre-processing step we binarize the training section (2–21) of the treebank with $h = 1$, $v = 2$ markovization (i.e., horizontal context limited to a single sibling, one vertical parent), left-factored. Aside from the fast tree kernel just discussed, we also test with a re-implementation of the quadratic tree kernel, to compare the effects of the choice of programming language and representation. The results are in table 2.

	Time (hr:min)		
Implementation	CPU	Wall clock	# fragments
Sangati et al. (2010): Quadratic tree kernel	160	10:00	1,023,092
This work: Quadratic tree kernel	93	6:15	1,032,568
This work: Fast tree kernel	2.3	0:09	1,023,880

Table 2: Performance comparison. Wall clock time is when using 16 cores.

Our quadratic tree kernel gets a modest 1.7 fold speedup, which is probably due to the more low-level style of programming. The real gain comes from the asymptotic speedup of the fast tree kernel: we obtain a 70-fold speedup over `FragmentSeeker`; 40-fold over our quadratic tree kernel.

The results show some variation in the number of fragments found. To validate our results, we compare the output of our system to that of `FragmentSeeker`. If the output of the latter is taken as a gold standard, and we disregard frequencies, we get an F_1 score of 99.93 % for our implementation of the fast tree kernel; i.e., there are only about 1500 fragments over which there is disagreement.

If exact frequencies[2] are computed the fast tree kernel needs 13 minutes; the frequencies match exactly with those of `FragmentSeeker`. With `FragmentSeeker` exact frequencies require another 5 hours to compute. For each fragment, the whole treebank is traversed to count its occurrences, which is again quadratic but with larger constant factors due to the number of nodes in recurring fragments. We need only 4 minutes by employing an index of the sets of trees containing a particular production. By taking the intersection of the sets for the productions in a fragment, we get a precise list of candidate trees which could contain the fragment one or more times. The actual number of occurrences is then counted by traversing these trees.

The source code of our implementation is available for download as part of `disco-dop`, at `https://github.com/andreasvc/disco-dop`

Acknowledgements

Thanks to Federico Sangati for making the source code of his `FragmentSeeker` available.

2 Approximate frequencies are returned by default, which are the frequencies of extracting a fragment as maximal fragment of a pair of trees, while exact frequencies include all occurrences.

References

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

Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330.

Moschitti, Alessandro (2006). Making tree kernels practical for natural language learning. In *Proceedings of EACL*, pages 113–120. Available from: `http://acl.ldc.upenn.edu/E/E06/E06-1015.pdf`.

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