LASSY WORK PACKAGE 6

CASE STUDY 1
AUTOMATIC EXTRACTION OF HYPERNYMY INFORMATION

Deliverable 6.1

Erik Tjong Kim Sang
Alfa-informatica
University of Groningen

LASSY WP 6.1: LASSY Tools: Case Study I

Erik Tjong Kim Sang
Alfa-informatica
University of Groningen

October 31, 2009

1 Introduction

We describe the use of the LASSY toolkit for extracting evidence for noun hypernymy relations from a syntactically annotated Dutch corpus. The results of this experiment have been discussed in Tjong Kim Sang and Hofmann (2009). In this report, we provide a detailed description of the steps which were required to extract the relevant data from the corpus.

Section two outlines the linguistic problem that we would like to solve. Section three presents the steps executed with the toolkit for retrieving the required data from the annotated corpus. In section four we summarize the experiment results. The final section contains some concluding remarks.

2 Hypernymy

Our goal is to extract evidence for a hypernymy relation between nouns from a text corpus. Hypernymy is a relation between two words which expresses that one word covers the meaning of the other while at the same time having a broader meaning. For example, *color* is a a hypernym of *red* and *animal* is a hypernym of *cat*.

Evidence for the presence of a hypernymy relation can be found in text. For example, the phrase *colors such as red* indicates that *color* is a hypernym of *red*. By using a seed list of pairs of related words, interesting phrases like this one can easily be extracted from a text collection. If the text collection contains part-of-speech information, we can then find new pairs of related words by looking for other instances of, for example, *Noun such as Noun*.

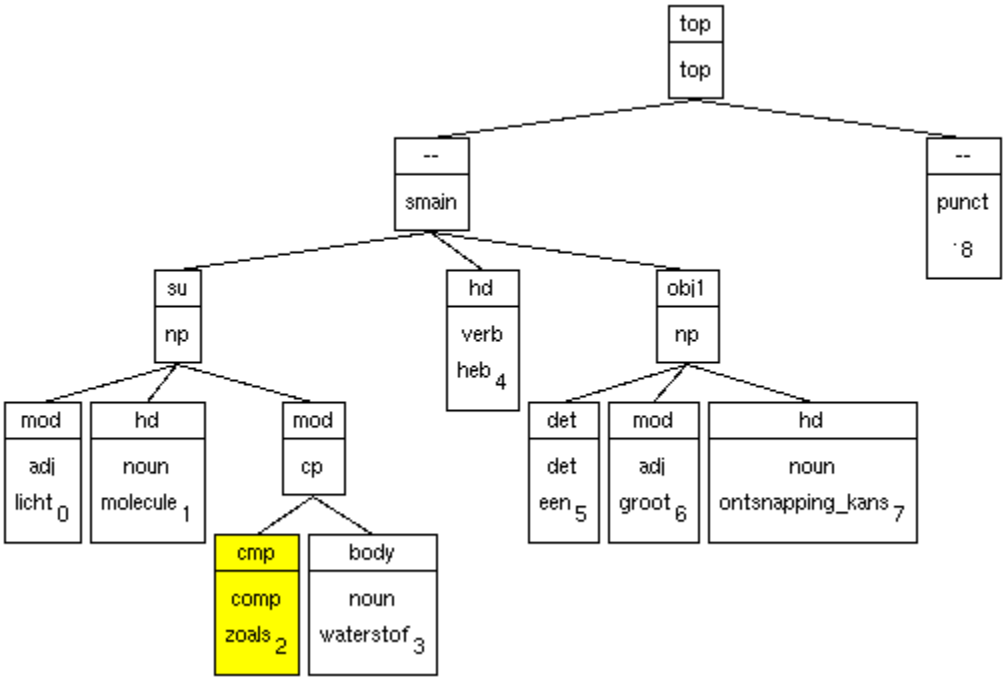

Figure 1: Tree representation of the syntactic annotation of sentence 32-64-3 of the LASSY corpus containing the phrase *light molecules such as hydrogen* (*lichte moleculen zoals waterstof*). The phrase suggests that *molecule* is a hypernym of *hydrogen*.

However, using only part-of-speech information in text patterns would lead to incorrect classifications, like suggesting *book* as a hypernym for *O'Reilly* from the phrase *publishers of books such as O'Reilly*. In orde to minimize such errors, we will look for text patterns in a syntactically annotated corpus. This will enable us to look for noun phrase heads linked by the phrase *such as*.

An example of a syntactically analyzed sentence containing hypernymy information, can be found in Figure 1. The phrase *light molecules such as hydrogen* (*lichte moleculen zoals waterstof*) is evidence for a possible hypernymy relation between *molecule* and *hydrogen*. Other candidate noun pairs for the hypernymy relation can be found by looking for the same phrase with different nouns. In that case we could use the syntactic annotation shown in Figure 1 as an additional clue and look for noun phrases with a modifier (mod) with head *zoals* (*such as*) and a noun as body. Additional modifiers of the candidate nouns, like adjectives, can be used for building more precise alternative text patterns for finding related words. In fact, we would like this example phrase to produce two identification patterns:

Noun1 su - ; zoals Mod Noun1 ; Noun2 Body zoals
licht mod Noun1 ; Noun1 su - ; zoals Mod Noun1 ; Noun2 Body zoals

Here the arcs of the tree have been converted to syntactic dependency relations between head words

of nodes ($word_1$ Relation $word_2$). The lexical information of the target nouns (Noun1 and Noun2) and out-of-phrase words (-) has been removed in order to obtain general patterns which fit other noun pairs. The dependency triples have been ordered from left to right, in the same order as an imaginary path connecting the two target nouns. One alternative extraction pattern was built using the modifier information in the phrase, the adjective *licht* (*light*). The pattern is shown at the second line.

This example phrase generates just two of many candidate environments that could be used as evidence for the hypernymy relation. There are several variants involving this phrase, for example with or without commas, with noun phrases or bare nouns which may include several additional modifiers. The goal of this case study is to extract from a corpus all possible syntactical contexts (paths shorter than some maximum length) that could express a hypernymy relation between two nouns. An extra statistical analysis will estimate how well each of these contexts predicts the relation by keeping track of the presence of known related word pairs associated with a context. A machine learner will be used for building a reliable predictor from this information.

3 Application of the LASSY Toolkit

In this section we present a step-by-step procedure for retrieving from syntactically annotated corpora the linguistic data required for our hypernymy experiment. We start with describing the installation of Saxon XQuery processor (section 3.1). Then we present an initial version of the query for retrieving the required information (section 3.2). In section 3.3 we show how the path between two nodes can be computed. Section 3.5 discusses working with large data sets. The final complete query can be found in Appendix B.

3.1 Installing an XQuery retrieval environment

We use syntactically annotated data from the LASSY corpus [Van Noord, 2009]. The annotations have been encoded in XML. An example of the data annotation can be found in Figure 2. In order to retrieve the relevant information for our hypernymy task from the corpus, we need a language for searching in XML documents. We have selected XQuery as the most appropriate language for this task [Tjong Kim Sang, 2009].

In order to be able to apply XQuery statements to the available annotated text, we need a environment which can interpret XQuery commands. For this purpose, we use the Saxon XQuery processor. We downloaded the open-source Java version from http://saxon.sourceforge.net/ and created a new directory in which we unzipped the source file saxonb9-1-0-7j.zip. This operation produced nine jar files and two directories, one with documentation and one with notices.

As a test we ran a query which retrieves all lexical information from the sentences in the text corpus and returns it between `<s>` tags:

```
<alpino_ds version="1.2">
 <node cat="np" rel="su">
  <node pos="adj" rel="mod" root="licht" word="Lichte"/>
   <node pos="noun" rel="hd" root="molecule" word="moleculen"/>
   <node cat="cp" rel="mod">
     <node pos="comp" rel="cmp" root="zoals" word="zoals"/>
     <node pos="noun" rel="body" root="waterstof" word="waterstof"/>
   </node>
 </node>
 ...
 <sentence>Lichte moleculen zoals waterstof ...</sentence>
</alpino_ds>
```

Figure 2: XML code representing the hypernymy context *Light molecules such as hydrogen* from sentence 32-64-3 of the Wikipedia section of the LASSY corpus. Words can be found in the `word` attributes of the tag `node` while annotation information is stored in other attributes of the tag. The complete annotated sentence can be found in Appendix A.

```
for $node in //alpino_ds
return <s>{$node//sentence/text()}</s>
```

The query searches for all `alpino_ds` elements, each of which contain a sentence in the annotated documents. The words of the sentence are stored in `sentence` elements. The query returns the text that can be found in these elements, enclosed between <s> and </s> tags.

We store this query in a separate file `query.xq` and enter the following command on the command line for sending the query to the Saxon XQuery processor:

```
java -cp saxon9.jar net.sf.saxon.Query -s 32-64-3.xml -q:query.xq
```

This command applies the query that is stored in the file `query.xq` (option -q) to the data file 32-64-3.xml (option -s), which contains a single annotated sentence from the Wikipedia section of the LASSY corpus. It requires the availability of Java (we used version 1.5.0.12). The command's output contains the words of the sentence:

```
<?xml version="1.0" encoding="UTF-8"?>
<s>Lichte moleculen zoals waterstof hebben een grotere ontsnappingskans .</s>
```

This completes the installation test of the XQuery interpreting. We are now ready for building a more complex program.

3.2 Initial query in XQuery

The goal of this case study is to extract arbitrary syntactic contexts containing two nouns. One sentence may generate several of such syntactic contexts and one word can appear in different contexts. We want to extract relevant context information for each candidate word. This requirement prevents us from using the LASSY tool dtview for selecting the required information since it can only select a single context related to a single candidate word. We need to use a programming environment like XQuery in order to identify and output the syntactic context of two arbitrary nouns (cf. Tjong Kim Sang (2009), section 3.1).

Figure 2 shows the XML annotation structures which corresponds to the hypernymy context *light molecules such as hydrogen*. There are three relevant basic parts which we would like to extract from the phrase: the two head nouns that are candidates for the relation (*molecules* and *hydrogen*) and the words between them (*such as*). Additionally, we are also interested in words modifying the nouns (in this case only the adjective *light*). The following pseudocode can be used for retrieving this information:

```
1 FOR node1 IN sentence-nodes, node2 IN sentence-nodes
2    compute path between nodes
3    IF node1 CONTAINS noun AND
4       node2 CONTAINS noun AND
5       node1 BEFORE node2 AND
6       path SHORTER THAN maximum-length
7    THEN
8       PRINT node1, node2, path-between-nodes
```

Selecting nodes from a set of nodes (pseudo code line 1), selecting noun nodes (lines 3 and 4), checking the order of two nodes (5) and printing information (8) can be translated to XQuery commands in a straightforward way. Computing the path between two nodes is mode complicated. For this task we will write a separate XQuery function.

Here is the core XQuery code associated with the query pseudocode:

```
for $node1 in //node, $node2 in //node
(: path computation skipped for now :)
where  $node1/@pos = "noun" and (: require $node1 to contain a noun :)
       $node2/@pos = "noun" and (: require $node1 to contain a noun :)
       (: require words of $node1 to precede those of $node2 :)
       $node1/@end cast as xs:integer <= $node2/@begin cast as xs:integer
       (: path length comparison skipped :)
return <node>
         <pair word1="{$node1/@word}" word2="{$node2/@word} />
```

```
      <path/>
    </node>
```

It chooses two nodes `$node1` and `$node2` from the syntactic tree, checks if they contain a part-of-speech attribute with value noun and also if the first word is located before the second, and then outputs the two words. The output for the document 32-64-3.xml from the Wikipedia section of the LASSY corpus looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
   <pair word1="moleculen" word2="waterstof"/>
   <path/>
</node>
<node>
   <pair word1="moleculen" word2="ontsnappingskans"/>
   <path/>
</node>
<node>
   <pair word1="waterstof" word2="ontsnappingskans"/>
   <path/>
</node>
```

The input sentence contains three nouns (see Figure 2). Each of them is combined with the other two to create three candidate pairs. Path information is missing from the output data. The next section discusses a query extension which extracts this missing information.

3.3 Extension: retrieving a dependency path

A dependency path consists of the nodes in the shortest path between the nodes that have the target words as head word. For example, for the first word pair in the previous output example, *moleculen – waterstof* the path would contain three nodes (compare with Figure 2):

1. a node with relation tag `su`, category tag `np` and head word *molecules*
2. a node with relation tag `mod`, category tag `cp` and head word *zoals*
3. a node with relation tag `body`, pos tag `noun` and head word *waterstof*

We will represent the path as a sequence of node pairs: 1-2 and 2-3. We extend our extraction query by creating a new function which retrieves this path information and by adding a call to the function to return part of the query:

```
let $path := local:path($node1,$node2)
...
<path>{$path}</path>
```

The new function `local:path` retrieves the ancestors of the two nodes, stores these in two lists, removes the duplicates from the two node lists and returns the results. The code looks like this:

```
declare function local:path($node1 as element(node), $node2 as element(node)) {
   (: use built-in function for retrieving ancestors :)
   let $ancestors1 := $node1/ancestor-or-self::node()
   let $ancestors2 := $node2/ancestor-or-self::node()
   (: remove duplicates from ancestor lists :)
   let $uniqueA1 := $ancestors1 except $ancestors2
   let $uniqueA2 := $ancestors2 except $ancestors1
   (: return results: relation codes and lexical information :)
   for $node in (reverse($uniqueA1),$uniqueA2)
      return(<pnode>{$node/(@rel,@word)}</pnode>)
};
```

Here is part of the output which the modified query produces for the example sentence shown in Figure 2:

```
<node>
   <pair word1="molecules" word2="waterstof"/>
   <path>
      <pnode word="molecules" rel="hd"/>
      <pnode rel="mod" word="zoals">
      <pnode rel="body" word="waterstof">
   </path>
</node>
```

We now have access to path information but not yet in the required format. We would like the path nodes (`pnode`) to contain information about pairs of nodes, for example <`pnode pword="zoals" rel="body" word="waterstof"`>, with information about the head words of the involved phrases. Some but not all heads are marked by relation attribute `hd`. We will use the Alpino library function `head-of` to obtain all head nodes:

```
(: load alpino library functions in namespace alpino :)
import module namespace alpino = "alpino.xq" at "alpino.xq";
(: in the function path :)
let $parentHead := alpino:head-of($parent)
```

```
(: load alpino library functions in namespace alpino :)
import module namespace alpino = "alpino.xq" at "alpino.xq";

declare function local:path($node1 as element(node), $node2 as element(node)) {
   (: use built-in function for retrieving ancestors :)
   let $ancestors1 := $node1/ancestor-or-self::node()
   let $ancestors2 := $node2/ancestor-or-self::node()
   (: remove duplicates from node lists :)
   let $uniqueA1 := $ancestors1 except $ancestors2
   let $uniqueA2 := $ancestors2 except $ancestors1
   (: return results: lexical and part-of-speech information and relations :)
   for $node in (reverse($uniqueA1),$uniqueA2)
   let $parentHead := alpino:head-of($node/..) (: get the parent's head :)
   let $nodeHead := alpino:head-of($node)      (: get the node's head :)
   where $nodeHead/@id != $parentHead/@id      (: skip self-links :)
   return(<pnode pword="{$parentHead/@word}"
           ppos="{$parentHead/@pos}">
           {$nodeHead/@rel,$nodeHead/(@pos,@word)}</pnode>)
};

for $node1 in //node, $node2 in //node
let $path := local:path($node1,$node2)
where  $node1/@pos = "noun" and
       $node2/@pos = "noun" and
       $node1/@end cast as xs:integer <= $node2/@begin cast as xs:integer and
       count($path) <= 5
return <node>
         <pair word1="{$node1/@word}" word2="{$node2/@word}"/>
         <path>{$path}</path>
       </node>
```

Figure 3: Query for retrieving pairs of nouns with dependency paths from a syntactically annotated corpus.

The library function will retrieve the lexical item below the head node of the parent. Note that we will need to apply `head-of` both to the parent and to the node itself because we cannot rely on the node containing any lexical information (for example, syntax node do not contain words themselves). The resulting head words can be used in the output of the query. The output will contain pairs of words at arbitrary distances. We are only interested in paths of lengths five and less. We can express this requirement by adding an extra constraint to the `where` part in main section of the code:

```
   $node1/@end cast as xs:integer <= $node2/@begin cast as xs:integer and
   count($path) <= 5 (: new: restricts the path length to 5 :)
return ...
```

The complete new version of the query can be found in Figure 3.

3.4 Adding satellite information

In the example in Section 2, we showed that we were interested in obtaining two different types of paths. The first is the shortest sequence of nodes connecting the two target nodes. The second contains this base path as well as additional information from the sister nodes of the two nodes. Snow et al. (2005) presents a nice example of the usefulness of this additional type of information, where two nouns are linked by a path containing the word *as* with the important clue word *such* being a sister of the first noun. This word is vital for the evidence provided by the syntactic context for the relation between the two nouns. The additional sister nodes are called satellites.

Simply listing all sister nodes of the two target nouns, is not enough. Care should be taken that the satellites are not already present in the path. Furthermore, the satellites of the two nouns need to be combined. If the first noun has one satellite A in the syntactic analysis and the second noun has one satellite B then four different paths should be generated: one with only the base path, and three with additional satellite information: on A, on B and on both A and B.

In order to keep the query simple, we have only incorporated the duplicity check and moved the task of combining base paths and satellite information to a later processing stage, after XQuery processing. In the target output format, the satellite information is added to the path information:

```
<node>
   <pair word1="molecules" word2="waterstof"/>
   <path>
      <pnode pword="molecules" ppos="noun" rel="mod" pos="comp" word="comp/>
      <pnode pword="zoals" ppos="noun" rel="body" pos="noun" word="waterstof"/>
      <satellites>
         <satellite1 rel="mod" pos="adj" word="licht"/>
      </satellites>
   </path>
</node>
```

The new `satellite` tag contains two different daughters. The satellites of the first noun are stored in `satellite1` tags while the satellites of the second noun are stored in `satellite2` tags (not present in this example). Adding this new information to the output requires several adaptations of the query. Extra `for` statements should check the sisters of both target nouns and store in variables those that are not already part of the base path. The base path should be stored in a variable as well in order to enable the `path` function to return both the base path and the satellite information. The maximum size of the path should be increased by one since the new path returned by the `path` function contains an extra node with satellite information:

```
(: collect ids of path nodes :)
let $ids1 := ( for $node in ($uniqueA1)
               return(alpino:head-of($node)/@id) )
let $basePath  := ( for $node in (reverse($uniqueA1),$uniqueA2)
                    ...
(: collect relevant satellite nodes :)
let $satellites1 := ( for $node in ($node1/../*)
                      let $nodeHead := alpino:head-of($node)
                      where empty(index-of(($ids1,$ids2),
                                  alpino:head-of($node)/@id))
                      return(<satellite1>{$node/@rel,
                             $nodeHead/(@pos,@word)}</satellite1>) )
...
(: return path and satellite information to main part of query :)
return($basePath,<satellites>{$satellites1,$satellites2}</satellites>)
...
(: include satellites node in count :)
count($path) <= 6
```

A version of the query which includes all these changes (and some others) can be found in Appendix B. Appendix C shows the output of the query applied to file 32-64-3.xml of the Wikipedia section of the LASSY corpus.

3.5 Working with large data collections

Up until now, we have applied the query to a single document. In a real-life situation, queries will be applied to thousands if not millions of documents. In order to do this, both the query and the method for accessing the documents might need to be changed.

In the LASSY corpus, each corpus sentence is stored in a different file and thousands of these files are concatenated and compressed in files with extension .dz. We wrote a special program for decompressing these files and converting them to a format which is suitable for the Saxon XQuery processor (`dz2saxon`, see Appendix D). Note that the LASSY toolkit also includes a client-server architecture for accessing large collections of corpus files. This could be more suitable for other users but we did not test it.

Here is an example call of the `dz2saxon` program:

```
dz2saxon wik_part0001.data.dz wik_part0002.data.dz |
   java -cp saxon9.jar net.sf.saxon.Query -s - -q:query.xq
```

`dz2saxon` reads two collection files with Wikipedia sentences, and converts them. The results are

sent to Saxon. The program removes all but the first <?xml lines from the collection files and encloses every other material in a new top node element <corpus>. Rather than a file name argument, the Saxon call contains a hyphen after the option -s to indicate that the input should be read from standard input which in this case is the output of the script dz2saxon.

The new input files contain many sentences and we should take care that the only pairs of nodes with information from the same sentence are checked. This requires modification of the query:

```
for $sentence in /corpus/alpino_ds,
    $node1 in $sentence//node,
    $node2 in $sentence//node
```

A new variable $sentence has been introduced. It selects a sentence and $node1 and $node2 will be nodes from the tree associated with this sentence.

The only additional adaption which we had to make was increasing the amount of memory that could be used by Saxon (option -Xmx1999m):

```
dz2saxon wik_part001?.data.dz |
   java -Xmx1999m -cp saxon9.jar net.sf.saxon.Query -s - -q:query.xq
```

The final version of the query program can be found in appendix B. Note that it is not able to process an arbitrary number of sentences. In our experiment, it could handle 100,000 sentences but not 500,000 (because of a lack of available memory). For larger number of sentences, the query will need to be run several times on manageable numbers of sentences.

4 Experiment results

We applied the extraction query to two sections of the LASSY corpus [Van Noord, 2009]: a 2006 version of the Dutch Wikipedia (5 million sentences) and Dutch Newspapers (26 million sentences). The query output was processed by additional software which linked each pair to the relevant satellites and selected from the word pairs the ones that we were interested in (nouns from the Dutch section of EuroWordNet). This process resulted in 43 million combinations of noun pairs, paths and satellites from Wikipedia while the count for the newspapers was 470 million.

Next, the path information and the satellites were converted to numeric features with each feature representing a combination of a path with associated satellites (3,000 for Wikipedia and 13,000 for the newspapers). All word pairs were classified as either belonging to the hypernymy relation or not based on the information in the Dutch section of EurWordNet [Vossen, 1998]. This resulted in

Data source	Related	Precision	Recall	$F_{\beta=1}$
AD	706	42.9%	30.2%	35.4
NRC	1224	26.2%	25.3%	25.7
Parool	584	31.2%	23.8%	27.0
Trouw	760	35.3%	29.0%	31.8
Volkskrant	1204	29.2%	25.5%	27.2
Newspapers	3806	20.7%	29.1%	24.2
Wikipedia	1580	61.9%	47.0%	53.4

Table 1: Performance of hypernym noun pair extraction from the LASSY corpus. The data was divided in six sections: five Dutch newspapers (AD, NRC, Parool, Trouw and Volkskrant) and the Dutch Wikipedia. We measured the number of related pairs in each section and the precision, recall and $F_{beta=1}$ rates of the task of identifying these related pairs.

38,000 (Wikipedia) and 396,000 (newspapers) sets of features each with a binary class indicating whether the set corresponded with a hypernym pair or not.

The two data sets were randomly divided in 10 sections. The data was processed by the machine learning algorithm Bayesian Logistic Regression in 10-fold cross validation fashion: using each of the 10 sections as test data in turn while training on the other nine. The results were evaluated by measuring precision and recall. The evaluation scores can be found in Table 1.

The most interesting observation which can be made from the result table is the difference between the newspapers section and Wikipedia. The latter contains proportionally more related pairs while allowing the the machine learner to perform better. We suspect that the performance difference is caused by a larger number of repeated phrases in Wikipedia as well as a higher percentage of definition sentences.

One motivation for this study was to find reliable methods which can be used for automatically deriving lexical information. The results in Table 1 show that there is some room for improvement. For example, by using the Wikipedia data we found about 1200 candidate pairs of which 743 were correct. The precision score of 61.9% indicates that the pairs found by the query still need to be checked before they can be added to a lexical resource. Furthermore, the number of 743 correctly identified pairs (about 2% of the size of the Dutch section of EuroWordNet) was smaller than we had hoped for.

However, there is still good news. We have performed a parallel study in which we refrained from using dependency structure annotation and only used lexical and part-of-speech tag information [Tjong Kim Sang and Hofmann, 2009]. The performance levels reached by the extraction query in the two studies were almost identical. However the query produced more positive cases from dependency information than from lexical material. We were able to show that in order to neutralize the positive effect of syntactic annotation, the lexical approach would need to have access to 43% more data [Tjong Kim Sang and Hofmann, 2009]. Having access to syntactic annotations had a

positive effect.

5 Concluding remarks

This report has presented a case study on how syntactic information can be extracted from the syntactically annotated LASSY corpora. The description includes presentations of the target data, installation of the query processor and a step-by-step construction of the required query. Additionally, the report contains both an example input file and the output of the query for this example file.

The presented method has been able to derive hypernymy information. The quality and the quantity of the results was lower than we had hoped for. However, we have been able to show that the extraction method has benefited from having access to syntactic annotations. We have shown that a method relying only on lexical and part-of-speech information would need 43% more data in order to be able to find the same number of related word pairs [Tjong Kim Sang and Hofmann, 2009]. Our extraction method has benefited from having access to the dependency parses in the LASSY corpus.

References

[Snow et al., 2005] Snow, R., Jurafsky, D., and Ng, A. Y. (2005). Learning syntactic patterns for automatic hypernym discovery. In *NIPS 2005*. Vancouver, Canada.

[Tjong Kim Sang, 2009] Tjong Kim Sang, E. (2009). *LASSY WP 5.2: Specification of a Search and Extraction Tool*. LASSY project report, University of Groningen.

[Tjong Kim Sang and Hofmann, 2009] Tjong Kim Sang, E. and Hofmann, K. (2009). Lexical patterns or dependency patterns: Which is better for hypernym extraction? In *Proceedings of CoNLL-2009*. Boulder, CO, USA.

[Van Noord, 2009] Van Noord, G. (2009). Huge parsed corpora in lassy. In *Proceedings of TLT7*. LOT, Groningen, The Netherlands.

[Vossen, 1998] Vossen, P. (1998). *EuroWordNet: A Multilingual Database with Lexical Semantic Networks*. Kluwer Academic Publisher.

A Sentence 32-64-3 from the Wikipedia part of the LASSY corpus

The full syntactic annotation of the test sentence used for the examples in this report: sentence 32-64-3 of the Wikipedia section of the LASSY corpus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<alpino_ds version="1.2">
  <node begin="0" cat="top" end="9" id="0" rel="top">
    <node begin="0" cat="smain" end="8" id="1" rel="--">
      <node begin="0" cat="np" end="4" id="2" rel="su">
        <node begin="0" end="1" frame="adjective(e)" id="3" infl="e" lcat="ap" pos="adj"
         rel="mod" root="licht" sense="licht" word="Lichte"/>
        <node begin="1" end="2" frame="noun(both,count,pl)" gen="both" id="4" lcat="np"
         num="pl" pos="noun" rel="hd" root="molecule" sense="molecule" word="moleculen"/>
        <node begin="2" cat="cp" end="4" id="5" rel="mod">
          <node begin="2" end="3" frame="complementizer(zoals)" id="6" lcat="cp"
           pos="comp" rel="cmp" root="zoals" sc="zoals" sense="zoals" word="zoals"/>
          <node begin="3" end="4" frame="noun(de,mass,sg)" gen="de" id="7" lcat="np"
           num="sg" pos="noun" rel="body" root="waterstof" sense="waterstof"
           word="waterstof"/>
        </node>
      </node>
      <node begin="4" end="5" frame="verb(hebben,pl,transitive_ndev)" id="8" infl="pl"
       lcat="smain" pos="verb" rel="hd" root="heb" sc="transitive_ndev" sense="heb"
       word="hebben"/>
      <node begin="5" cat="np" end="8" id="9" rel="obj1">
        <node begin="5" end="6" frame="determiner(een)" id="10" infl="een"
         lcat="detp" pos="det" rel="det" root="een" sense="een"
         word="een"/>
        <node begin="6" end="7" frame="adjective(ere)" id="11" infl="ere" lcat="ap"
         pos="adj" rel="mod" root="groot" sense="groot" word="grotere"/>
        <node begin="7" end="8" frame="noun(de,count,sg)" gen="de" id="12" lcat="np"
         num="sg" pos="noun" rel="hd" root="ontsnapping_kans" sense="ontsnapping_kans"
         word="ontsnappingskans"/>
      </node>
    </node>
    <node begin="8" end="9" frame="punct(punt)" id="13" lcat="punct" pos="punct"
     rel="--" root="." sense="." special="punt" word="."/>
  </node>
  <sentence>Lichte moleculen zoals waterstof hebben een grotere ontsnappingskans
   .</sentence>
  <comments>
    <comment>Q#32-64-3|Lichte moleculen zoals waterstof hebben een grotere
     ontsnappingskans .|1|1|-0.08847426960247112</comment>
  </comments>
</alpino_ds>
```

B XQuery program

This is the final version of the XQuery program extracting from a syntactically annotated corpus pairs of nouns and the intermediate paths.

```
(: load alpino library functions in namespace alpino :)
import module namespace alpino = "alpino.xq" at "alpino.xq";

declare function local:path($node1 as element(node), $node2 as element(node)) {
   (: use built-in function for retrieving ancestors :)
   let $ancestors1 := $node1/ancestor-or-self::node()
   let $ancestors2 := $node2/ancestor-or-self::node()
   (: remove duplicates from node lists :)
   let $uniqueA1 := $ancestors1 except $ancestors2
   let $uniqueA2 := $ancestors2 except $ancestors1
   (: get ids :)
   let $ids1 := ( for $node in ($uniqueA1)
                  return(alpino:head-of($node)/@id) )
   let $ids2 := ( for $node in ($uniqueA2)
                  return(alpino:head-of($node)/@id) )
   (: dependency path information: lexical, part-of-speech and relations :)
   let $basePath  := ( for $node in (reverse($uniqueA1),$uniqueA2)
                       let $parentHead := alpino:head-of($node/..)
                       let $nodeHead := alpino:head-of($node)
                       where $nodeHead/@id != $parentHead/@id
                       return(<pnode pword="{$parentHead/@word}"
                                     ppos="{$parentHead/@pos}">
                       {$nodeHead/@rel,$nodeHead/(@pos,@word)}</pnode>) )
   (: satellite information: sisters of $node1 and $node2 :)
   let $satellites1 := ( for $node in ($node1/../*)
                         let $nodeHead := alpino:head-of($node)
                         where empty(index-of(($ids1,$ids2),
                                     alpino:head-of($node)/@id))
                         return(<satellite1>{$node/@rel,
                                $nodeHead/(@pos,@word)}</satellite1>) )
   let $satellites2 := ( for $node in ($node2/../*)
                         let $nodeHead := alpino:head-of($node)
                         where empty(index-of(($ids1,$ids2),
                                     alpino:head-of($node)/@id))
                         return(<satellite2>{$node/@rel,
                                $nodeHead/(@pos,@word)}</satellite2>) )
   return($basePath,<satellites>{$satellites1,$satellites2}</satellites>)
};

for $sentence in /corpus/alpino_ds,
    $node1 in $sentence//node,
    $node2 in $sentence//node
let $path := local:path($node1,$node2)
where  $node1/@pos = "noun" and
```

```
       $node2/@pos = "noun" and
       $node1/@end cast as xs:integer <= $node2/@begin cast as xs:integer and
       count($path) <= 6
return <node>
         <pair word1="{$node1/@word}" word2="{$node2/@word}" />
         <path>{$path}</path>
       </node>
```

C Output of XQuery program

This is the output of the XQuery program shown in Appendix B when applied to sentence 32-64-3 of the Wikipedia section of the LASSY corpus. For this task we adapted the program by removing the phrase `/corpus` which was added in order to enable processing collections of annotated sentences. The query was processed by the Saxon XQuery engine by starting the following command:

```
java -cp saxon9.jar net.sf.saxon.Query -s 32-64-3.xml -q:query.xq
```

The output was a list of word pairs defined in `pair` environments with additional intermediate paths encoded in `pnode` tags in `path` environments. Each path node had information of the two linked nodes (their head words and their part-of-speech class) as well as the syntactic relation between the two nodes. The output looked like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
   <pair word1="moleculen" word2="waterstof"/>
   <path>
      <pnode pword="moleculen" ppos="noun" rel="cmp" pos="comp" rword="zoals"/>
      <pnode pword="zoals" ppos="comp" rel="body" pos="noun" word="waterstof"/>
      <satellites>
         <satellite1 rel="mod" pos="adj" word="Lichte"/>
      </satellites>
   </path>
</node>
<node>
   <pair word1="moleculen" word2="ontsnappingskans"/>
   <path>
      <pnode pword="hebben" ppos="verb" rel="hd" pos="noun" word="moleculen"/>
      <pnode pword="hebben" ppos="verb" rel="hd" pos="noun" word="ontsnappingskans"/>
      <satellites>
         <satellite1 rel="mod" pos="adj" word="Lichte"/>
         <satellite1 rel="mod" pos="comp" word="zoals"/>
         <satellite2 rel="det" pos="det" word="een"/>
         <satellite2 rel="mod" pos="adj" word="grotere"/>
      </satellites>
```

```
   </path>
</node>
<node>
   <pair word1="waterstof" word2="ontsnappingskans"/>
   <path>
      <pnode pword="zoals" ppos="comp" rel="body" pos="noun" word="waterstof"/>
      <pnode pword="moleculen" ppos="noun" rel="cmp" pos="comp" word="zoals"/>
      <pnode pword="hebben" ppos="verb" rel="hd" pos="noun" word="moleculen"/>
      <pnode pword="hebben" ppos="verb" rel="hd" pos="noun" word="ontsnappingskans"/>
      <satellites>
         <satellite2 rel="det" pos="det" word="een"/>
         <satellite2 rel="mod" pos="adj" word="grotere"/>
      </satellites>
   </path>
```

D Compressed file conversion

For the conversion of the compressed annotated LASSY corpus files to a format that the Saxon XQuery processor can handle, we have used the Perl script `dz2saxon`. It decompresses the corpus files, removes lines containing `<?xml` and encloses the file contents in `<corpus>` tags. The call of the program should precede the call to Saxon for applying the query, for example:

```
dz2saxon wik_part001?.data.dz |
   java -Xmx1999m -cp saxon9.jar net.sf.saxon.Query -s - -q:query.xq
```

Here the program is applied to 10 input files, `wik_part0010.data.dz` to `wik_part0019.data.dz`, and the results are sent to Saxon.

This is the Perl code of `dz2saxon`:

```
#!/usr/bin/perl
# dz2saxon: unpack .dz file(s) and convert them to a format suitable for Saxon
# usage: dz2saxon file [files]
# 20090806 erikt(at)xs4all.nl

use strict;

my $command = $0;
my @files = @ARGV;
my $xml = "";
foreach my $file (@files) {
   open(INFILE,"gunzip -c $file |") or die "$command: cannot open $file\n";
   while (<INFILE>) {
```

```
      my $line = $_;
      chomp($line);
      if ($line !~ /<\?xml/) { print "$line\n"; }
      else {
         if ($xml eq "") {
            print "$line\n<corpus>\n";
            $xml = $line;
         }
      }
   }
   close(INFILE);
}
if ($xml ne "") { print "</corpus>\n"; }
exit(0);
```