RC23397 (W0411-102) November 10, 2004
Computer Science

IBM Research Report

Word Sense Disambiguation in a Slot Grammar Framework

Michael C. Mc Cord
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Word Sense Disambiguation in a Slot Grammar Framework

Michael C. McCord
IBM T. J. Watson Research Center

November 11, 2004

1 Introduction

This is a preliminary report on a system for word sense disambiguation (WSD) for unrestricted vocabulary, which requires no training on tagged text.[1] Disambiguation is done to WordNet word senses (Miller, 1995; Fellbaum, 1998).

The "disambiguating power" of the system comes from three sources:

- Parsing by English Slot Grammar (ESG) (McCord, 1980; McCord, 1990; McCord, 1993), both at training time and at runtime.
- The WordNet relation system, applied at training time.
- The WordNet sense frequency data (number of tagged senses), applied at runtime.

These resources are used as follows. For training, we do the following two things:

- ESG is used to parse a large (training) corpus, producing a *slot-filler database*, as described in Section 2. Keys of entries in the database are head-slot-filler tuples (and their inverses). The values are frequencies of occurrence of the keys in the parses of the corpus. These tuples serve as the basis for local context in the WSD.

[1]Except in the sense that it uses the tagged frequency information that comes with WordNet.

- Using the slot-filler database, along with WordNet relations, we form the *sense discriminator database* for the corpus, as described in Section 3. Keys of entries in this database are again head-slot-filler tuples (and their inverses), and the value of a key is the list of possible WordNet senses of the *index* of the key (its first component – a filler word or a head word), where each sense is paired with a "sense evidence number" for that sense of the index in the context of the slot-filler tuple.

At runtime, the sense discriminator database, as well as ESG and WordNet again, are used as follows by the WSD algorithm:

- Each sentence is parsed with ESG, and the parse is used in two ways:
    1. The slot-filler relations given by the parse are converted into keys for the sense discriminator database, and look-up of these keys allows us to accumulate *contextual evidence* for each possible sense of each word.
    2. Filtering of possible senses for each word is done through ESG's morphological analysis (especially for parts of speech). Also, ESG's analysis of phrasal verbs plays a role in deciding what WordNet entries to look up. For example, in "They brought most of the prices down", we know to try "bring down" in WordNet.
- The WordNet sense frequency data (for the word forms and parts of speech determined by ESG) are used to feed into a WSD score that also uses the contextual evidence for senses.

The runtime WSD algorithm is described in detail in Section 4. The algorithm can be applied in either *sentence* mode or *document* mode. In the former, senses are chosen only on the basis of local evidence from each sentence. In the latter, evidence is gathered across a whole document, with an assumption that each word is used in only one sense. (We plan to experiment with in-between versions of these.)

Using Slot Grammar (SG) slot-filler relations to express sentence-level context provides much more relevant information for WSD than n-gram methods, we believe, because slot-filling deals more directly with arguments of word sense predicates. Also, SG slot-filling can even be done *remotely*, as with *wh*-extraposition, or *logically*, as with passives and implicit subjects.

One reason this report is a preliminary one is that I have not done enough exploration of the literature in WSD, and I am not attempting in this first version to relate adequately to previous work. There are relations to the work in (Lin, 1997; Mihalcea and Moldovan, 1999; Mihalcea and Faruque, 2004; Leacock and Chodorow, 1998; Leacock et al., 1998; Yarowsky, 1995) and more, but I will add this discussion in a later version.

At this point, we are doing WSD only for common nouns, verbs and adjectives.

Section 5 describes an experiment with this system, and an evaluation of the performance. In this experiment, the training set consisted of about 1.7 million sentences from *Wall Street Journal* (WSJ) text. On a blind test set from the WSJ, the system performed at 72% accuracy. This was in sentence mode, because the test set sentences were not from a single WSJ article, but were spread out uniformly across a larger test set. It seems likely to me that the score would increase with training on a suitable larger corpus. The exact method of scoring is described in Section 5.

A secondary purpose of this report is to describe a kind of integration of WordNet with ESG, and this is presented in Section 6. This involves a new system of very fast API functions for WordNet which I wrote to do this work, along with an efficient editor-based WordNet browser which allows easy exploration of WordNet by clicking on synsets or words in the result displays of queries.

2 The Slot-Filler Database

The Slot Grammar package has a facility called **SlotStat** for processing a very large corpus and producing a *slot-filler database* associated with the corpus. The sentences of the corpus are parsed, and instances of *slot-filler tuples* occurring in the parses are gathered, along with their frequencies of occurrence in the corpus. See (McCord, 1993) and (Dagan and Itai, 1990) for earlier versions of these ideas.

Let us look first at an example before we explain the situation generally. Consider the sentence in (1), with its ESG parse.

```
(1) The treasurer deposited money in the bank.
```

```
  ┌─ndet        the1(1)             det sg def the ingdet
┌─•─subj(n)     treasurer1(2,u)     noun cn sg h
•───top         deposit1(3,2,4,5,u) verb vfin vpast sg vsubj
├───obj(n)      money1(4)           noun cn sg
└─•─comp(p)     in1(5,7)            prep staticp
  │ ┌ndet       the1(6)             det sg def the ingdet
  └─•─objprep(n) bank1(7,u)         noun cn sg
```

If sentence (1) is taken as the whole corpus, then SlotStat produces the following slot-filler database:

```
bank<comp<in<deposit        1
deposit<~comp<in<bank       1
deposit<~obj<n<money        1
deposit<~subj<n<treasurer   1
money<obj<n<deposit         1
treasurer<subj<n<deposit    1
```

On each line is a slot-filler tuple, followed by its frequency in the corpus. The character `<` separates the four elements of each slot-filler tuple.

The second element of each tuple is its slot (like `subj` for the subject of a clause), but the slot may be preceded by `~`, in which case we call it an *inverse slot*. When the slot is not an inverse, the first element of the tuple is the (citation form of the) head word of the filler phrase for the slot, and the last element is the head word of the matrix phrase. When the slot is an inverse, these first and last elements are reversed. But in both cases, we call the first element the *index word* of the tuple and the last element the *environment word*.

Notice that we cover only selected slots. A rough idea is that we include the complement slots for verbs, nouns and adjectives, plus two adjunct noun premodifier slots (called `nadj` and `nnoun`) whose main fillers are adjectives and nouns, respectively. We will not describe here the exact list of slots covered.

Whether the slot is inverted or not, the third element of a slot-filler tuple is the *option* for the slot chosen by the parser. In general, each slot of SG has associated *options*, which are symbols that describe or control, in part, how the slot is filled. For complement slots, the associated options come from

the lexical entry for the matrix word of the slot. In the following lexical entry for `buy` (simplified)

```
buy < v obj (iobj n for)
```

the `iobj` (indirect object) slot is shown with options `n` and `for`. The `n` option means that the `iobj` can be filled by an NP (as in *buy me a book*), and the `for` option means that the `iobj` can be filled alternatively by a `for`-PP (as in *buy a book for me*). The `obj` (direct object) slot in this example shows no option overtly, but the default is that `obj` has an `n` option if no option is specified.

In an SG parse tree, like the one in (1), the option chosen by the parser for each slot is shown in parentheses after the slot, as in `subj(n)`.

In the slot-filler tuples of a slot-filler database, there is special treatment for certain options, associated with "promotion" of certain slot-fillers. In parse (1), note that the filler of the slot+option `comp(p)`, a complement slot of the verb `deposit`, is actually the PP headed by `in`. So on that account, the associated (non-inverted) slot-filler tuple would be:

```
in<comp<p<deposit        1
```

But SlotStat modifies this by *promoting* the object of the preposition `in`, namely `bank`, to become the index word, and in the process replacing the actual option, `p`, of `comp(p)` by the preposition `in`, giving:

```
bank<comp<in<deposit        1
```

In this way, a direct relationship (important for WSD) between the content words `deposit` and `bank` is packed into a single tuple, and the identification of the preposition (`in`) is there too.

This promotion is done in some cases besides PP fillers as well. For instance it is done with `to`-VPs, for which ESG views `to` as the head. The verb head of the actual VP is taken as the filler, and a remnant of `to` is coded in the option (`inf`) of the tuple. This is also done with `that`-clauses.

There is another, slightly different, kind of promotion done in creating the slot-filler database. This has to do with verb-particle constructions and certain cases of verbs with PP complements. Let us look first at particles.

Consider the example in (2).

(2) The company phased those plans out.

```
 ┌─ndet     the1(1)          det sg def the ingdet
┌•─subj(n)  company1(2,u)    noun cn sg
•──top      phase1(3,2,5,6) verb vfin vpast sg vsubj
│┌─ndet     those1(4)        det pl def
└•─obj(n)   plan1(5,u)       noun cn pl
└──comp(pt) out1(6,u)        prep motionp badobjping
```

Particles associated with verbs are treated by ESG as complement slot fillers – for the slot `comp` – with the option being either `pt` (used with specific particles listed in the lexical entry) or `lo` (standing for "locational") which takes a range of particles and PPs which can have an interpretation as physical location (but may be used abstractly also). This treatment of verb particles as ordinary verb slot fillers makes it easy to have them separated from the verb, as in example (2).

For example (2), the slot-filler database produced by SlotStat is:

```
company<subj<n<phase:out   1
phase:out<~obj<n<plan      1
phase:out<~subj<n<company  1
plan<obj<n<phase:out       1
```

So when there is a verb particle complement `P` present for a verb `V`, the particle is concatenated the verb like so, `V:P`, in the slot-filler tuple.

It is worth storing the particles with the verbs in the slot-filler database for two reasons. (1) This packs more information into each slot-filler tuple. (2) WordNet often stores verb+particles as multiword strings consisting of the verb and particle with a blank separator ("phase out" is an example), especially when the verb+particle has a non-compositional sense. So it is convenient for us to store sense information directly with the combination `V:P`. When this is looked up in WordNet, we of course replace the colon by a blank. But in the slot-filler database, it is useful to use the colon instead of a blank, because some verb multiwords in the ESG lexicon are expressed with blank separators for their components.

The second case of promotion of this sort for the slot-filler database is done for certain cases of verbs with PP complements. For these we write `V#P`, where `V` is the verb and `P` is the preposition that heads the PP complement. An example is `call#for`. As with some verb+particle constructions,

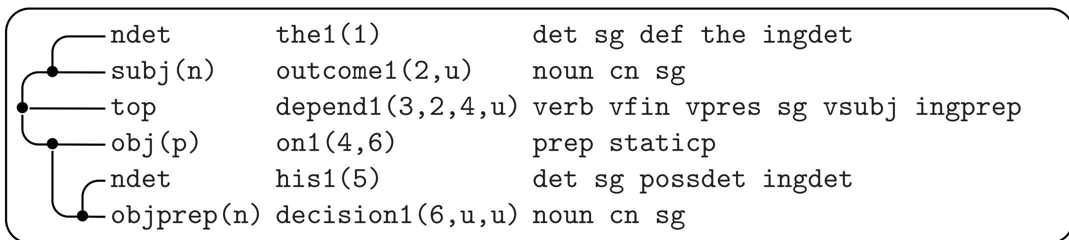
WordNet also sometimes stores senses under such verb+preposition multiwords, for example "call for".

We do `V#P` promotion only for restricted kinds of occurrences of PP complements. For instance we do *not* do it in the case of "deposit in" in example (1), even though the `in`-PP is a complement of `deposit`. We avoid it in this case because an NP direct object (`money`) is present. In general, we have elected to create a `V#P` promotion in the following two cases, where the general idea is to try to capture situations where the phrasal verb `V#P` is more likely to have its own special meaning, with a separate listing for `V P` in WordNet.

1. The verb `V` has a complement filling an `obj` (direct object) slot with option allowing a PP headed by `P`. Examples are "depend on" and "believe in". In this case, after the promotion to `V#P` in the slot-filler tuples, we replace the option of the `obj` slot by `n` (for NP), and let the filler be the object of the preposition `P`.
2. The verb `V` has a complement filling a `comp` slot with option allowing a PP headed by `P`, and `V` has no direct object slot filled (logically) in the sentence. Examples are "build on" and "dip into". In these two examples, the verb frame *could* in general also have an `obj` slot filled, but we do the promotion only if there is no `obj` logically filled. In a promotion of this sort, in the slot-filler tuples we replace the `comp` slot by `obj` and let its option be `n`.

An example for Case 1 is shown in (3).

(3) `The outcome depends on his decision.`



The slot-filler database produced for (3) is:

```
decision<obj<n<depend#on   1
depend#on<~obj<n<decision  1
depend#on<~subj<n<outcome  1
outcome<subj<n<depend#on   1
```

An example for Case 2 is shown in (4).

(4) `He only dipped into the report.`

```
subj(n)     he(1)           noun pron sg def nom h perspron
vadv        only1(2)        adv post ppadv nounadv
top         dip1(3,1,u,4)   verb vfin vpast sg vsubj
comp(p)     into1(4,6)      prep motionp
ndet        the1(5)         det sg def the ingdet
objprep(n)  report1(6,u)    noun cn sg
```

The slot-filler database produced for (4) is:

```
dip#into<~obj<n<report  1
dip#into<~subj<n<he     1
he<subj<n<dip#into      1
report<obj<n<dip#into   1
```

When we create the sense discriminator database from the slot-filler database, as described in the next section, we will need to look up the WordNet senses of promoted forms like `V#P`. Our candidate senses will be obtained from look-up of *both* the multiword form `V P` *and* the simple verb `V`. This is appropriate because some of the meanings of `V P` may in fact depend only on `V` and are listed in WordNet only under `V`. For instance, the sense of “dip into” in “He dipped into the pool” is listed (appropriately) under “dip” in WordNet 2.0, and the only sense listed under the multiword “dip into” is “read selectively from”.

There are some cases where one finds appropriate senses for `V:P` (the verb-particle case) under `V` as well as under `V P`, analogously to what we do for `V#P`. But from what I have seen so far, it seems best to stick with `V P` for the verb-particle case. I plan to investigate this more.

Let us now summarize, and describe the general form of slot-filler tuples. Each such object is a quadruple of the following form:

```
IndexWord<Slot<Option<EnvironmentWord
```

The `Slot` is either an inverse slot (preceded by `~`) or an ordinary slot. If it is an ordinary slot, then the `IndexWord` is the (head word of the) filler of the slot, or a promoted filler as described above, and the `EnvironmentWord` is the matrix word for the slot. For an inverse slot, these first and last members

of the tuple are reversed. The `Option` is by default the slot's chosen option, but may provide other information, such as a specific preposition, in the case of a promoted filler.

The slot-filler database created by SlotStat for a corpus consists of the slot-filler tuples extracted from parses of segments in the corpus, each paired with its frequency (total number of occurrences) in the corpus. During the corpus analysis, SlotStat stores the accumulating database in a hash table where the keys are the slot-filler tuples, and the values are frequency numbers. At the end of the corpus analysis, SlotStat stores the results in a file in a way that is convenient for reading back in. Results can be accumulated across several runs of SlotStat.

The reader will no doubt note that the inverse forms of slot-filler tuples and the associated frequency values are immediately obtainable from their non-inverted forms, so there is a kind of redundancy there. In fact SlotStat can optionally work with only the non-inverted forms, and this is the normal way of first creating the database. However, when we pass to a sense discriminator database (described in the next section), where the keys are also slot-filler tuples, we actually need to use both inverted and non-inverted forms. The symmetry is lost. This happens because the value for each key-tuple is associated specifically with the index word (first element) of the tuple, and has to do with sense disambiguation of that word.

Display (5) shows a listing of entries in a slot-filler database for a specific word, `paper`. These represent the most frequent tuples with index word `paper`, down to a certain frequency, in the database described in Section 5 below.

(5)

```
paper<~nnoun<n<court         171
paper<~nadj<a<international  157
paper<nnoun<n<product        142
paper<~nadj<n<num            107
paper<nnoun<n<work           105
paper<obj<n<file             103
paper<nnoun<n<company        96
paper<subj<n<say             70
paper<nnoun<n<industry       55
paper<obj<n<make             46
paper<nnoun<n<maker          45
paper<~nadj<a<local          43
```

```
paper<obj<n<use              42
paper<~nadj<a<new            42
paper<~nadj<a<other          40
paper<obj<n<coat             39
paper<subj<n<have            39
paper<nnoun<n<stock          36
paper<~nadj<en<coat          35
paper<nnoun<n<concern        34
paper<obj<n<publish          27
paper<obj<n<sell             27
paper<~nadj<n<year           27
paper<obj<n<buy              22
paper<obj<n<close            22
paper<~nadj<a<light          22
paper<nnoun<n<machine        21
paper<nobj<for<demand        21
paper<~nnoun<n<eurodollar    21
paper<nnoun<n<producer       20
paper<obj<n<issue            20
paper<obj<n<read             20
paper<~nadj<a<daily          19
paper<~nnoun<n<research      19
paper<nnoun<n<making         18
paper<obj<n<produce          18
paper<obj<n<sign             17
paper<obj<n<write            17
paper<nnoun<n<manufacturer   16
```

3 The Sense Discriminator Database

Next we use the slot-filler database for a corpus to construct the *sense discriminator database* for that corpus, where the key new information comes from WordNet.

The entries of a sense discriminator database are of the form:

`IndWord<Slot<Option<EnvWord` $\quad$ Sense$_1$-Evidence$_1$ Sense$_2$-Evidence$_2$ ...

The keys, `IndWord<Slot<Option<EnvWord`, are slot-filler tuples, just as for a slot-filler database. In the value for such a key, shown here after the blanks,

the $\texttt{Sense}_i$ are integers that specify the WordNet senses of `IndWord` with the part of speech it must have according to the slot and option shown for it in the tuple. These synset numbers are special to my own API to WordNet. Each corresponding $\texttt{Evidence}_i$ is an integer that measures the *evidence* for $\texttt{Sense}_i$, obtained as described below.

We build these entries as follows. For each entry

```
IndWord<Slot<Option<EnvWord  Freq
```

in the slot-filler database, and for each possible sense $\texttt{Sense}_i$ of `IndWord` (for its given part of speech), we determine $\texttt{Evidence}_i$ as follows. We first construct the set C of *closely related words* to `IndWord` under sense $\texttt{Sense}_i$ (conceptually) in two steps.

The first step is to form the set S of *closely related senses* to the pair (`IndWord`, $\texttt{Sense}_i$) by traversing WordNet away from $\texttt{Sense}_i$ via the following six relations:

1. **identity** (just arrive at $\texttt{Sense}_i$ itself)
2. **hypernym** (`@`)
3. **hyponym** (`~`)
4. **verb group** (`$`) if $\texttt{Sense}_i$ is a verb
5. **similar to** (`&`) if $\texttt{Sense}_i$ is an adjective
6. **also see** (`^`) if $\texttt{Sense}_i$ is a verb or adjective

For all of these, we move at most one step away from $\texttt{Sense}_i$, except for **hyponym**, for which we try up to two steps. The last of these relations, **also see**, is lexical and depends on the word `IndWord` as well as its sense $\texttt{Sense}_i$, but all the rest depend only on $\texttt{Sense}_i$. We let the set S of senses consist of all the senses (synsets) we arrive at from $\texttt{Sense}_i$ by these transitions.

The second step in forming the set C of closely related words to `IndWord` is to take the synsets in S, and for each such synset, add all its synonym members to C – excluding `IndWord` itself if it is present.

Actually, we put a limit on the number of words that can be added to C; this limit is currently set to 50. The steps for forming S and C have an order – starting with $\texttt{Sense}_i$ and then applying the six relations above in the order listed. In actuality, we add to C immediately after applying each of the six relations. And we stop the process when we reach the limit (currently 50).

Given C, we compute `Evidence`$_i$ as `Freq` plus the sum of all frequencies `F` where `CloseWord` is a member of C, and

```
CloseWord<Slot<Option<EnvWord  F
```

is in the slot-filler database.

Display (6) shows some of the entries for the index word `paper`, taken from the sense discriminator database described in Section 5.

(6)

```
paper<comp<of<make        77910-16 33615-5 32928-5 32942-5 32865-5 42740-5 20582-11
paper<~nadj<en<recycle    77910-13 33615-6 32928-6 32942-6 32865-6 42740-6 20582-6
paper<obj<n<write         77910-17 33615-84 32928-17 32942-83 32865-17 42740-17 20582-17
paper<obj<n<sell          77910-94 33615-27 32928-59 32942-27 32865-27 42740-59 20582-571
paper<~nnoun<n<government 77910-26 33615-288 32928-26 32942-9 32865-9 42740-26 20582-31
paper<~nadj<a<local       77910-142 33615-55 32928-167 32942-43 32865-43 42740-142 20582-156
paper<~nadj<a<large       77910-44 33615-9 32928-51 32942-9 32865-9 42740-106 20582-60
```

The WordNet sense numbers involved here are shown in the following output from the ESG+WSD interface to WordNet, where the hypernyms are given only to one level:

```
1. <77910> n: paper
  a material made of cellulose pulp derived mainly from wood or rags or certain grasses
  ==> <75790> material, stuff
2. <33615> n: composition, paper, report, theme
  an essay (especially one written as an assignment)
  ==> <33614> essay
  ==> <3897> written assignment, writing assignment
3. <32928> n: newspaper, paper
  a daily or weekly publication on folded sheets
  ==> <32907> press, public press
4. <32942> n: paper
  a scholarly article describing the results of observations or stating hypotheses
  ==> <32934> article
5. <32865> n: paper
  medium for written communication;
  ==> <32861> medium
6. <42740> n: newspaper, paper, newspaper publisher
  a business firm that publishes newspapers
  ==> <42738> publisher, publishing house, publishing firm, publishing company
7. <20582> n: newspaper, paper
  a newspaper as a physical object
  ==> <21670> product, production
```

4 The Algorithm for WSD

The idea is as follows. For each node `N` (having one of our selected parts of speech) of the ESG parse tree, the *possible senses* of `N` are the the WordNet senses of `N` having the part of speech given for `N` by ESG. For each possible sense of node `N`, we gather the *evidence* (a real number) for that sense of `N`. Then we choose a sense that has the maximum evidence. The WordNet senses of a word are ordered, and if several senses have the same maximum evidence, we choose the first one.

The algorithm can be run in two modes – (a) *sentence* mode or (b) *document* mode. For mode (a), the evidence gathering and sense determinations are done on a segment-by-segment basis. Mode (b) runs in two passes over a document (or selected sequence of segments). In the first pass, the evidence is gathered, with an assumption that each word-POS pair is used in only one sense in the given text. Then in the second pass, senses are chosen as indicated in the preceding paragraph. We plan to experiment with variations, where the evidence is gathered for certain windows around segments.

In our current system, the evidence for a sense has two components:

1. the *contextual* evidence
2. the *sense frequency* evidence.

The contextual evidence comes from slot-filler relations in the ESG parse tree, plus the sense discriminator database, in a way that we will explain below. The total evidence for a sense is a linear combination of these two components, and currently we have found that we get the best results with a weighting of 0.46 for the contextual evidence and 0.54 for sense frequency evidence.

The sense frequency evidence for a sense of a node `N` is obtained from the sense tagging counts provided by WordNet for the senses of each word. We just normalize these counts by dividing the count for each possible sense of node `N` by the sum of the counts for all the possible senses of `N` (if this sum is greater than 0).

Let us now explain the determination of the contextual evidence for a sense of a node `N`. The basic idea is to look at the various slot-filler relations in which `N` participates in the parse tree, and to accumulate evidence for each possible sense of `N` by adding up the evidence numbers associated with those slot-filler relations in the sense discriminator database. For instance, suppose we have:

```
Smith deposited money in a savings bank.
```

Then `bank` participates in the relations:

```
bank<comp<in<deposit
bank<~nnoun<n<savings
```

Then we look up these relations in the sense discriminator database, and, for each noun sense of `bank` we add up the two associated evidence numbers.

So, in general, suppose we are given a node `N` (of one of our selected parts of speech) in the parse tree, and a possible sense `S` of `N`. What is the contextual evidence `E` of `S` as the appropriate sense? We initialize `E` to 0. Let `CiteN` be the citation form of `N`. Then for each slot-filler relation in the parse tree of the form

```
CiteN<Slot<Option<EnvWord
```

we look up this relation in the sense discriminator database, and if it is there, we find the evidence number `E1` associated with sense `S`, and we increment `E` by `E1`. We do this for each possible sense `S` of `N`, and then normalize the associated evidence numbers by dividing each by the sum of all these evidence numbers for the possible senses of `N` (if the sum is non-zero).

5 Experiment and Evaluation

The training and testing for the experiment were done on text from the *Wall Street Journal* (WSJ). "Training" almost exclusively means producing the slot-filler database by running SlotStat on the training set, and from that, producing a corresponding sense discriminator database. As indicated above, training does not include any sense tagging or annotation.

The main part of the training set consisted of 1.67M segments of WSJ text. The rest of the training set, as well as the test set, came from the WSJ portion of the Penn Treebank (PTB), which has about 44K segments. I had earlier used this part of the PTB for some measurements of ESG parsing accuracy, and split it 75% by 25% into a training set and a blind test set. There was very little actual training of ESG on this training set – mainly only some lexical extraction. For the current ESG+WSD evaluation, I added this PTB training set (the unannotated text segments only) to the training set, making the total number of training segments about 1.7M. I

selected the ESG+WSD test set automatically from my PTB test set (again, unannotated text segments only), choosing 100 segments by taking every n^{th} segment, where n is chosen to make the total 100 segments. I have not checked how related (in period of the WSJ) the 1.67M-segment WSJ training text is to the PTB WSJ text.

The 100 test segments were annotated with correct WordNet senses, with the aid of some tools that I developed to go with ESG+WSD and a text editor (Kedit so far). The tools make it quick to see the available WordNet senses for a word that one clicks on, and then to import the sense marking one selects into the annotated file. A program in ESG+WSD can read an annotated corpus of this sort and dynamically score ESG's performance on it. Some segments from the training set were also annotated, for scoring feedback in experiments with the WSD algorithm.

Example (7) below shows a segment from the test set along with its sense annotation table.

The first column of the table consists of the words (or multiwords) of the segment in citation form, each with an associated node number. The node number is just the segment position number the word, or of the head word in the case of a multiword.

The second column contains in each row the marked sense (synset) number of the chosen word sense. This is a WordNet sense number in my API to WordNet, described briefly in Section 6. If WordNet has no appropriate sense for the given word in context, then it is annotated with a sense `<-1>`. This happened with only 12 out of 877 words in the test set (of our chosen parts of speech). When a word is annotated with `<-1>`, the scoring program ignores that word in the tallying of good and bad.

The third and fourth columns are redundant, because they are implied by the sense number and WordNet, but they are there for the convenience of a human reader. They are ignored by the scoring program. The third column shows the part of speech, and the fourth shows the definition, of each sense (as supplied by WordNet).

```
(7) Omnicare said it expects the division to add "substantial
sales volume and to make a positive contribution to our earnings
in 1990 and beyond."
```

```
Omnicare(1)
say(2)              <84455> v: express in words
it(3)
expect(4)           <83136> v: regard something as probable or likely
the(5)
division(6)         <43444> n: an administrative unit in government or business
to(7)
add(8)              <80512> v: make an addition (to)
substantial(9)      <96708> adj: fairly large
sale(10)            <5523> n: the general activity of selling
volume(11)          <27520> n: the property of something that is great in magnitude
and(12)
to(13)
make(14)            <92156> v: engage in
a(15)
positive(16)        <93598> adj: involving advantage or good
contribution(17)    <3852> n: any one of a number of individual efforts in a common endeavor
to(18)
our(19)
earnings(20)        <68678> n: the excess of revenues over outlays in a given period of time
in(21)
1990(22)
and(23)
beyond(24)
```

I ran SlotStat on the 1.7M-segment training set, and used an option that outputs only slot-filler tuples whose frequency is at least a specified minimum. I chose a minimum of 5. This produced approximately 533K slot-filler tuples (in symmetric form), together with their frequencies. Samples are given in example (5) in Section 2. Next I ran the ESG+WSD utility for creating the sense discriminator database, and this produced approximately 487K entries. There are fewer than for the slot-filler database, because some of entries in the latter have index words with no WordNet senses (this being the case mainly for proper nouns). Samples for this sense discriminator database are given in (6) in Section 3.

Then I ran ESG+WSD on the test set, using this sense discriminator database. It is actually run (in this case) under the management of a scoring function operating on the gold sense-annotated test set (consisting of segments, each followed by its gold sense annotation table.) In this mode of running, the scoring function gives ESG+WSD each (simple, unannotated)

text segment to parse and do its WSD on. ESG+WSD produces its own internal dynamic version of the sense annotation table for the segment. The scoring function also reads the gold sense annotation table listed after the segment, and compares this with what ESG+WSD produced.

The comparison and scoring proceed as follows. For each segment, the scoring function goes through the nodes of the segment and tallies up two things:

1. A count `SN` ("Sense Nodes") of nodes whose senses we are considering. These are just the nodes in the gold sense table that have a sense label `<nnn>` marked on them, except for those marked `<-1>` (no appropriate WordNet sense). As indicated above, the nodes marked with `<nnn>` (including `<-1>`) should be the nodes for common nouns, verbs and adjectives.
2. A count `GSN` ("Good Sense Nodes"), where ESG+WSD agrees with the gold table marking. To *agree*, (a) the ESG+WSD node and the gold table node must either be both marked or be both unmarked, and (b) if they are both marked, they must have the same marking. ESG+WSD never marks a node with `<-1>`, so if the gold table has this, there will be no agreement; but then such nodes are not counted in the `SN` tally either.

The counts `SN` and `GSN` are accumulated (starting with 0) over the whole test set, and the score on the test set is the ratio `GSN/SN` as a percentage.

In the whole process, the scoring function can produce a report file, showing both the gold sense tables, and for each, what ESG+WSD produced, expressed in the same format. "Local" scores for each segment are given (based on the `SN` and `GSN` tallies just for that segment), as well the total score at the end.

Example (8) shows the ESG+WSD output, along with its score, for the segment given in (7) from the gold file.

```
(8) Omnicare said it expects the division to add "substantial
sales volume and to make a positive contribution to our earnings
in 1990 and beyond."
```

```
Omnicare(1)
say(2)             <84455> v: express in words
it(3)
expect(4)          <83136> v: regard something as probable or likely
the(5)
division(6)        <43395> n: an army unit large enough to sustain combat
to(7)
add(8)             <80512> v: make an addition (to)
substantial(9)     <96708> adj: fairly large
sale(10)           <5523> n: the general activity of selling
volume(11)         <71591> n: the amount of 3-dimensional space occupied by an object
and(12)
to(13)
make(14)           <92156> v: engage in
a(15)
positive(16)       <103524> adj: characterized by or displaying affirmation or acceptance ...
contribution(17)   <3852> n: any one of a number of individual efforts in a common endeavor
to(18)
our(19)
earnings(20)       <68678> n: the excess of revenues over outlays in a given period of time
in(21)
1990(22)
and(23)
beyond(24)
```

```
Sense score = 72.73%
```

For this 100-segment test set, the score produced by ESG+WSD was 72%.

The scoring procedure we have described here can do the WSD in sentence mode (with sense evidence gathering done on each segment in isolation), or it can apply the WSD algorithm in the document mode (with two passes) that we described above in Section 4. The experiment we just described was done in sentence mode, because the test segments are widely dispersed in the PTB corpus. We plan to do experiments also in document mode.

6 Integration of ESG and WordNet

To do this work, I developed my own set of API functions (in C) to WordNet, along with an internal data structure representation of WordNet, in such a way that these are integrated nicely with the Slot Grammar (SG) code and framework. Basically, ESG is just packaged with these extra functions as an add-on. SG is coded in pure ANSI C and runs efficiently on several platforms. The SG WordNet API functions are quite fast, mainly because the chosen data representation stores all of WordNet in memory in arrays, and there is maximum use of (internal, binary) integer values in the arrays, which in turn index into other arrays.

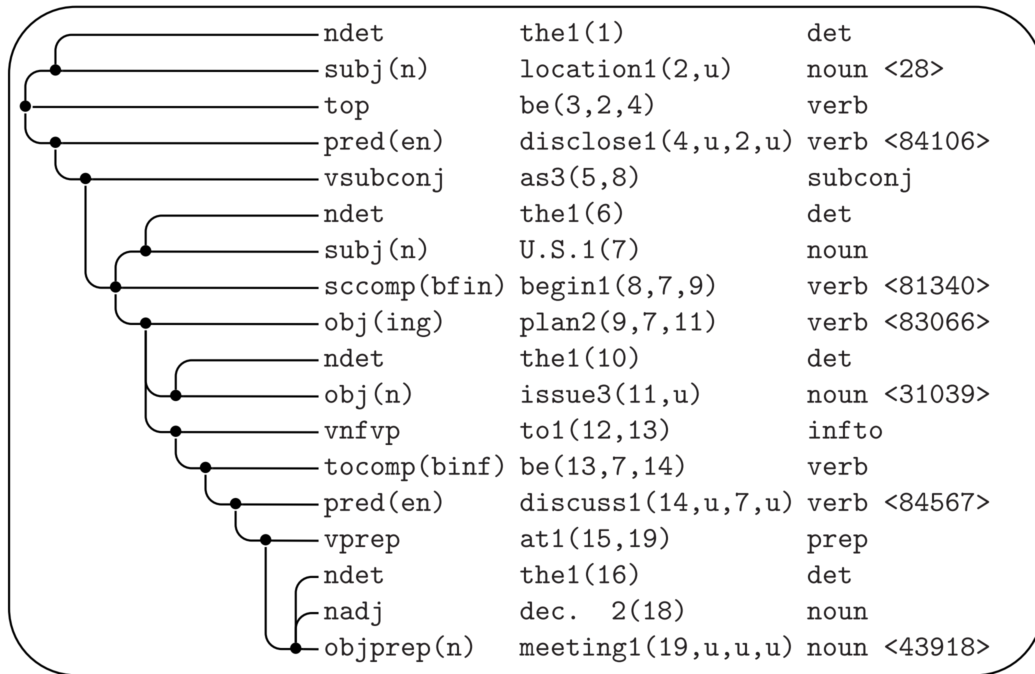
To use ESG+WordNet with a version of WordNet depends only on the lexical *data files* distributed with WordNet, not on any of the code. ESG commands (given to the ESG executable) can "compile" the standard distribution WordNet data files (from about any version of WordNet) to transmogrified data file versions of them which can be read into memory (into the special internal representation) very quickly when ESG initializes. This compilation (needed only once) is very quick – taking about 4 seconds. And the ESG initialization plus loading of these WordNet files takes about a fourth of a second.

Any users of this package would be expected to obtain WordNet themselves directly from Princeton, and then use the ESG-based compiling operation just discussed.

The work described in this report was done with WordNet 2.0. The sense/synset numbers exhibited are from the ESG-based data representation. They are obtained simply by starting with 0 for the first noun synset occurring in `noun.dat` and counting forward synset by synset. Then the indexing of synsets continues forward with `verb.dat` (without resetting the counter), then `adj.dat`, and `adv.dat`. All four of the synset families are stored in one array.

The SG data structure for parses was expanded to include a new field for the WordNet sense of each node chosen by WSD, represented simply as an integer, the synset number described in the preceding paragraph. Parse displays can optionally show these sense numbers. An example of a sentence and such a parse display is shown in (9). In order to make the parse tree fit the page better, we have suppressed the display of features besides parts of speech. The parse is not quite correct. The phrase for "to be discussed ..." should be attached to "issues" instead of to "planning".

(9) The location was disclosed as the U.S. began planning the issues to be discussed at the Dec. 2 meeting.



This sort of parse display can show synset definitions as well as synset numbers.

The ESG+WordNet package also includes a browser for WordNet. Most of the browser code is in C and is coded generically in the ESG+WordNet package. It is intended to work with ascii text editors like Kedit that have a nice macro language, and I have so far implemented the editor side of the browser only for Kedit, where a few macros are required. The browser allows one to explore WordNet through all of its links, displaying the results in a normal editor window for a normal (temporary) file created by the ESG+WordNet code. In the display of results, one can click on any synset or word and explore from there. ("Clicking" means putting the cursor at the appropriate place and pressing a designated function key.) The depth of searching (like *all* hyponyms, or 2 levels of hyponyms) can be set by available commands. One can also type in words and other data to pop-up boxes.

The Kedit browser screen looks like this:

```
KEDIT - [C:\clmt\wn.out]
File  Edit  Actions  Options  Window  Help
====>
<0> n: entity
  that which is perceived or known or inferred to have its own distinct existence (living o
  [hyponym]:
  ==> <1> n: thing
    ==> <2> n: anything
    ==> <3> n: something
    ==> <4> n: nothing, nonentity
    ==> <23642> n: subject, content, depicted object
    ==> <48012> n: body of water, water
      ==> <47942> n: backwater
      ==> <47960> n: bay
        ==> <47830> n: Abukir, Abukir Bay
        ==> <47876> n: Andaman Sea
        ==> <47961> n: Bay of Bengal
        ==> <47962> n: Bay of Biscay
        ==> <47963> n: Bay of Campeche
        ==> <47964> n: Bay of Fundy
        ==> <47965> n: Bay of Naples
        ==> <47986> n: bight
          ==> <47987> n: Bight of Benin
          ==> <48349> n: Great Australian Bight
        ==> <47994> n: Biscayne Bay
        ==> <48039> n: Buzzards Bay
        ==> <48058> n: Cape Cod Bay
        ==> <48103> n: Chesapeake Bay
F1=word F2=wordc F3=exit F4=hype F5=hypo F6=pholo F7=pmero F8=sholo F9=smero
F10=mholo F11=mmero C+F1=simto C+F2=atri C+F3=entail C+F4=cause C+F5=vgroup
C+F6=sa C+F7=memsa C+F8=reg C+F9=memreg C+F10=use C+F11=memuse C+F12=help
S+F1=anto S+F2=alsee S+F3=back S+F4=pertto S+F5=partvb S+F6=domrel S+F7=ranrel
A+F1=begword A+F2=endword A+F3=lev1srch A+F4=fullsrch A+F5=entlev A+F6=topyns
Line=15  Col=1  Alt=0,0;0  Size=74107  Files=1  Windows=1  INS  R/W  7:59 AM
```

The main area of the screen shows the results for whatever search command was given, and the grey portion at the bottom is a key to the commands and their associated function keys. (There is a help screen also.)

This particular screen shows the results of finding the complete hyponym tree for the synset **entity**. There are 74,107 lines in the file being edited. The total time for the tree to be computed by ESG+WordNet and for the result screen to come up is about a fourth of a second (on a ThinkPad).

In doing the WSD work described in this report, I augmented ESG's base lexicon with entries derived from WordNet. The ESG base lexicon has about 86,000 entries – head words in citation form. ESG does derivational and inflectional morphology with the base lexical entries, so that many more words are covered that way. But, not trying to take account of morphology, I extracted (automatically) words from WordNet which (a) do not occur in the ESG base lexicon and (b) have a noun sense in WordNet, and created a SG-style addendum lexicon from these. This has about 75,000 entries.

They are mainly multiwords or proper nouns (or both). One trick in the extraction is to specify the index components of multiwords (the component that gets inflected), because these have to be specified to ESG in order to handle inflections of multiwords. (But ESG assumes that the last component is the index word if no component is marked, and this is usually right for nouns.)

I keep this WordNet-based addendum lexicon separate from the ESG base lexicon, with the idea that it should always be derived automatically from any new updates of WordNet.

References

[Dagan and Itai, 1990] Ido Dagan and Alon Itai. 1990. Automatic acquisition of constraints for the resolution of anaphoric references and syntactic ambiguities. *Proceedings of Coling-90*, 3:162–167.

[Fellbaum, 1998] Christiane Fellbaum. 1998. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA.

[Leacock and Chodorow, 1998] Claudia Leacock and Martin Chodorow. 1998. Combining local context with WordNet similarity for word sense identification. In Christiane Fellbaum, editor, *WordNet: An Electronic Lexical Database*. MIT Press.

[Leacock et al., 1998] Claudia Leacock, Martin Chodorow, and George A. Miller. 1998. Using corpus statistics and WordNet relations for sense identification. *Computational Linguistics*, 24(1):147–165.

[Lin, 1997] Dekang Lin. 1997. Using syntactic dependency as local context to resolve word sense ambiguity. *Proceedings of the ACL*.

[McCord, 1980] Michael C. McCord. 1980. Slot Grammars. *Computational Linguistics*, 6:31–43.

[McCord, 1990] Michael C. McCord. 1990. Slot Grammar: A system for simpler construction of practical natural language grammars. In R. Studer, editor, *Natural Language and Logic: International Scientific Symposium, Lecture Notes in Computer Science*, pages 118–145. Springer Verlag, Berlin.

[McCord, 1993] Michael C. McCord. 1993. Heuristics for broad-coverage natural language parsing. In *Proceedings of the ARPA Human Language Technology Workshop*, pages 127–132. Morgan-Kaufmann.

[Mihalcea and Faruque, 2004] Rada Mihalcea and Esanul Faruque. 2004. SenseLearner: Minimally supervised word sense disambiguation for all words in open text. *Proceedings of SENSEVAL-3*.

[Mihalcea and Moldovan, 1999] Rada Mihalcea and Dan I. Moldovan. 1999. A method for word sense disambiguation of unrestricted text. *Proceedings of the ACL*.

[Miller, 1995] George Miller. 1995. WordNet: A lexical database. *Communications of the ACM*, 38:39–41.

[Snyder and Palmer, 2004] Benjamin Snyder and Martha Palmer. 2004. The English all-words task. *Proceedings of SENSEVAL-3*.

[Stevenson and Wilks, 2001] Mark Stevenson and Yorick Wilks. 2001. The interaction of knowledge sources in word sense disambiguation. *Computational Linguistics*, 27(3):321–349.

[Stevenson, 2003] Mark Stevenson. 2003. *Word Sense Disambiguation: The Case for Combining Knowledge Sources*. CSLI Publications, Stanford, CA.

[Yarowsky, 1995] David Yarowsky. 1995. Unsupervised word sense disambiguation rivaling supervised methods. *Proceedings of the ACL*.