Faculté de Philosophie et Lettres

Département de Langues et Littératures Modernes - Orientation Germaniques

MEASURING LEXICAL PROXIMITY

Promoteur académique

M. A. MICHIELS, chargé de cours

Travail de Fin d'Etudes présenté par
Kevin Noiroux
En vue de l'obtention du grade de Master en Langues et Littératures modernes, orientation germaniques, à finalité approfondie, orientation études computationnelles

Année académique 2008-2009

Table of Contents

Foreword

"You shall know a word by the company it keeps", tells Firth at the end of the 1950s, at the time where computers had just been invented. This is what the programme *Lexdis* aims at: computing lexical proximity between any two words, as long as they are nouns, verbs, adjectives or adverbs. To do so, *Lexdis* makes use of different types of lexical databases, i.e. dictionaries and thesauri such as Roget's Thesaurus, or more recently, WordNet.

To illustrate the concept of lexical proximity, consider the pair "mouse" and "radiator". These two words have absolutely nothing in common. In other words, their lexical proximity is low and accordingly, *Lexdis* should produce a very low number. Now, consider the pair "cat" and "dog". Their lexical proximity is clearly higher. *Lexdis* should produce a much higher number.

Lexdis is a Prolog programme written by A. Michiels, professor at the University of Liège. The English Department at The University of Liège has also made all the arrangements to acquire the databases which, later, were converted into Prolog files. My contribution lies in attempting to provide an explanation of the semantic and computational concepts that are used in *Lexdis* to compute lexical proximity; in documenting *Lexdis* itself and in trying to improve the weighting algorithm.

In chapter I, I introduce the programming language that was used to write *Lexdis*. In chapter II, I comment on the concepts of word sense and word sense disambiguation in computational linguistics. In chapter III, I describe how *Lexdis* works and in chapter IV the databases that it uses. In chapter V, I comment on the code of the programme. Finally, chapter VI contains the evaluation of *Lexdis*.

Chapter I: Short Introduction to Prolog

0. Introduction

This chapter aims at introducing basic computational concepts without which it would be hard to understand how *Lexdis* works. *Lexdis* is written in Prolog, a programming language rather different from usual programming languages such as Java, PHP or C for instance. The implementation of Prolog that was used is Swi-Prolog, which among other things has the advantage of being free.

1. Declarative Programming Language

There exist two types of programming languages, i.e. procedural and declarative. In order to reach a solution in a procedural language, the programmer has to think about how to solve the problem stage by stage, while in declarative programming his or her task is more related to the description of the conditions for a solution within a particular universe.

Consider a programme that tells whether the number that the user inputs is positive, negative or equals zero. The algorithm in any procedural language is very likely to sound like this: if the number is greater than zero, it is positive; if the number equals zero, than it is zero; else the number is negative. The Prolog algorithm is written in a different way:

```
number(X) :- X>0,
             write('positive').
number(0) :- write('zero').
number(X):- X<0,
            write('negative').
```

The three possibilities are described. The programme is going to try them one at a time, and as soon as one succeeds, it is done. Of course, it is possible to paraphrase a declarative algorithm into a procedural one, but it is impossible to write a procedural algorithm in a declarative style.

Obviously, Prolog's being a declarative programming language does not make it de facto better. Declarative and procedural languages both have their sets of features that make them more suited to solving different types of problems. A very powerful feature specific to declarative languages, at least to Prolog, is "backtracking", and I will comment on it later.

2. Terminology

I am going to use terms specific to Prolog. This section contains a brief definition of most of them.

A *constant* is a fixed expression in a programme. It is opposed to a *variable* which contains information that varies from one execution of the programme to the other. By convention in Prolog, variables start with a capital letter or an underscore. For instance, Prolog is a variable and prolog is a constant. Quoting a term automatically turns it into a constant; e.g. 'Prolog', 'Computational Linguistics' and 'computational linguistics' are all constants.

A *predicate* is defined by its *functor* and *arity*. The functor is the name of the predicate, the arity the number of arguments it takes. In the predicate timetable(Date, Hour, Class), the functor is timetable and its arity is three as it takes three arguments. It may be referred to as timetable/3. In devising a procedure, there is no limit to the number of arguments and no restriction on their type (integer, string, etc.). The functor must be written with a lower case, so that it is not construed as a variable. Finally, the predicate timetable/3 is different from, for instance, the predicate timetable/2 as they have a different arity.

A Prolog programme is made up of *clauses*. Clauses are either *facts* or *rules*. Facts are the usual way to represent data in Prolog. Consider the following programme that tells a widow or widower's name:

```
% facts
male('John').     female('Jane').
male('Jake').     female('Anna').
dead('Mike').     dead('Amy').
partner('Anna', 'Mike').
partner('Amy', 'Jake').

% rules
widow(Female)[1] :-[2]
                  female(Female),[3]
                  partner(Female, Male),[3]
                  dead(Male).[3]
widower(Male)[1] :-[2]
                  male(Male),[3]
                  partner(Female, Male),[3]
                  dead(Female).[3]
/* example queries
1 ?- widow(Who).
Who = 'Anna'.

2 ?- widower('Amy').
false.

3 ?- widower('Jake').
true. */
```

Facts are data that rules are going to use. Here, they specify who is male or female, who is dead and who is in a relationship. The rules are composed of the head of the clause ([1]), the neck of the clause ([2]) and one or more goals ([3]) separated by commas. The declarative formulation of the first clause is: a person is a widow if she is female and her partner is dead; the procedural formulation is: a person must be first female and then her partner must be dead in order to be a widow. In other words, the procedural reading emphasizes that the order in which the goals are satisfied is more important. Nevertheless, it is completely untrue that a declarative reading of a rule implies that "the order of the clauses defining a predicate and the order of the goals in the body of each rule are irrelevant" (Bramer 213). For instance, the widow-programme would be much more efficient with the dead-goal before the

gender-goal, since it is never going to select a living person in the first place. This is true whether it is read procedurally or declaratively. Finally, a predicate is defined by all the clauses that have the same functor and the same arity:

```
number(X) :- X>0,
                write('positive').
number(0) :- write('zero').
number(X):- X<0,
               write('negative').
```

The programme contains three clauses number/3, each of them handling a specific problem. These three clauses define the predicate number/3.

Unification is a process that entails verification and instantiation. Consider the following clause: partner(Female, Male).. The query partner(X). will fail because unification checks that the structures are compatible. The query partner('Jane', 'John'). will be satisfied because the structures match. Then, instantiation begins: the variables Female and Male are respectively bound with the constants 'Jane' and 'John'.

3. Data Structure: Lists

Algorithms manipulate data structures, the most powerful type of which is the list. The list is a compound term that can be used to handle any type of information. Consider the following lists:

```
[one, two, three, Variable].
[01-01-09, 'New Year',5].
[[1,2],[apple, banana]].
[one].
[]. % empty list
```

Syntactically, a list is made up of a number of elements separated by commas, the square brackets beginning and ending it. There is no limit to the number of elements and the list can be empty. A list may contain other lists. In Prolog, a list may contain different types of data (integer, string, etc.). What makes lists a very powerful tool, is that a lot of operations are already implemented in suitable libraries, i.e. a set of pre-written Prolog programmes. For instance, it is possible to sort, reverse and permute a list; two lists can be concatenated to get a new list; duplicates can be removed from a list, etc. Another very important feature of the list is that it can be split up into a head and a tail:

```
[Head|Tail].
[one | two, three, Variable].
        Head = one.
        Tail = [two, three, Variable].
[[1,2]|[apple, banana]].
        Head = [1,2].
        Tail = [apple, banana].
[one].
        Head = one.
        Tail = [].
[].
```

The vertical line (" | ") is used to split up a list. Unless the head of the list is already a separate list, it will always be a non-compound term. The tail is always a list. If the list contains only one element, the head is the element and the tail is the empty list. The empty list is the only list that cannot be broken into head and tail. The splitting up of a list is paramount to any data treatment of lists as it allows

```
[H1, H2 | Tail].
[one, two | three, Variable].
        H1 = one.
        H2 = two.
        Tail = [three, Variable].
```

recursion, which amounts to satisfying a goal on the head of the list, and then applying the same goal to its tail, i.e. to all the other elements.

4. Recursion and Loops

Recursion and *loops* are means that allow a goal to be executed a number of times that is not known prior to the execution of the programme. The two cases for iteration are the repetition of a goal an unknown number of times or the treatment of an unknown quantity of information. In procedural programming languages, loops and recursion are used separately, but in Prolog, it is very frequent to use recursion to loop. While a loop entails the repetition of a goal, recursion entails the repetition of the goal in the same goal. In other words, recursion is a type of loop. Consider the following programme that prints all the arguments of a list on the screen:

```
myprint([Head|Tail]) :-
                print(Head),
                myprint(Tail).

myprint([]).
```

This goal uses recursion to loop. It extracts the first argument of the list, prints it and executes itself with the rest of the list as an argument. The second clause is very important: when there is only one element left in the list, its tail is the empty list. As mentioned above, the empty list cannot be split up into head and tail. So the last clause makes sure the goal succeeds with an empty list even thought there is no goal connected with the head of the clause. This is a classic out-of-loop clause. The only loop that does not use recursion is the fail-loop. As it uses backtracking, I will comment on it in the next section.

5. Backtracking

Backtracking is "the process of going back to a previous goal to find alternative ways of satisfying it." (Bramer 210). Let's take another version of the widow-programme:

```
% facts
male('John').     female('Jane').
male('Jake').     female('Anna').
dead('Mike').     dead('Amy').
partner('Anna', 'Mike').
partner('Amy', 'Jake').

% rules
widow(Female) :-
        female(Female),
        partner(Female, Male),
        dead(Male).
```

Prolog backtracks because it fails – the goal is not satisfied – or because the user asks it to. Say you want to know all the widows' names. The query that does that is `widow(X).` or any other variable instead of `X`. First, Prolog unifies the variable `Person` with the variable `X`. Then it selects the first female, sets a choice flag and executes the next goal.

The problem is that the first female is Jane and she does not have a partner since there is no partner('Jane', Husband). clause. So Prolog fails, backtracks to the latest choice point, tries with another clause, in this case female(Anna). and sets another choice flag. This one succeeds. Nevertheless, it is possible to ask Prolog to look for other solutions by pressing the semicolon key[1] (" ; "), in which case Prolog backtracks as if it had failed. Backtracking is a very powerful feature as Prolog has no problem backtracking through a complex hierarchy of choice points. Nevertheless, it may be useful to prevent backtracking. Consider the following programme:

```
number(X) :- X>0,!,
                write('positive').
number(0) :- write('zero'), !.
number(X):- !,
               write('negative').
```

In this case, there is no point asking Prolog for other results since a number is either positive, negative or equals zero. This is why there are *cuts* (!), represented by exclamation marks. The cut discards all the previous choice points of the clause. Here, the cut has another function. The programme tests whether the number is positive, but does not test whether it is negative. It works on the assumption that if Prolog reaches the third predicate, the number must be negative inasmuch as the two previous goals failed. The cuts make sure Prolog cannot look for other solutions if the number is positive, because if it could, it would find that a positive number is negative.

Backtracking also allows fail-loops. Imagine you want facts to be printed successively:

```
% facts
female('Jane').   female('Julie').
female('Eve').    female('Anna').


% rule
myprint :-
        female(Fem),
        print(Fem),
        nl, % new line
        fail.
myprint.
```

The query is simply myprint.. Prolog picks the first female, sets a choice flag there, prints her name on the screen, prints a new line and fails. Since it has failed, it backtracks and picks the next female and the whole process starts again. When there is no female left, the first myprint clause fails and Prolog tries the next one which succeeds vacuously. The fail-loop is the only loop which does not use recursion.

6. Finding All Solutions to a Goal

As mentioned above, hitting the semicolon key after getting a solution allows Prolog to backtrack and give other solutions. So, it is possible to get all the solutions to a query. There

[1] The semicolon means "or" in Prolog.

exist three predicates that fill a list with all the possible solutions to a goal, the easiest to understand being findall/3:

```
% facts
male(John).       female(Jane).
male(Jake).       female(Anna).
female(Eve).      female(Julie).
dead(Jake).       dead(Anna).
dead(Julie).
% rule
widow(Person) :-
                  female(Person),
                  dead(Person).
```

The normal query that fetches or checks the name of a window is widow(X).; findall puts all the solutions in a list: findall(Person,widow(Person),Solutions).. The second argument is the goal, the first argument is the relevant variable of the second argument and finally, the third argument is the list that contains all the solutions. The two other predicates, setof/3 and bagof/3, work the same way, but behave differently on failure; setof/3 also sorts and removes duplicates from the resulting list.

With such predicates, a programme can be written to find just one solution. Apply findall/3 to the main goal and you get all the solutions. It is a very powerful feature that has no equivalent in procedural programming languages.

Chapter II: Word Senses and Word Sense Disambiguation in Computational Linguistics

1. Word Senses

1.0. Introduction

Very often, words have different meanings. Any of these meanings is a word sense. This phenomenon is called "polysemy": a word is polysemic if it has more than one meaning. Agirre and Edmonds make an interesting comment on the difference between polysemy and ambiguity:

> [Polysemy] is an intrinsic property of words (in isolation from text), whereas "ambiguity" is a property of text. Whenever there is uncertainty as to the meaning that a speaker or writer intends, there is ambiguity. So, polysemy indicates only potential ambiguity, and context works to remove ambiguity. (8)

For instance, "mouse" is polysemic: it refers either to an animal or to a computer device. If during a conversation it is not clear which one the speaker is referring to, there is ambiguity.

Distinguishing between word senses is far from an easy task. Indeed, the disambiguation of "mouse" belongs to the easiest cases. Most of the time, words senses overlap: it is hard to know where one sense ends and the next one begins. Consider the adjective "charming". Unlike "mouse", it does not display two clear meanings. Still, it is used in different contexts:

- You have a charming little house.
- Your friends are charming.

In both examples, "charming" has the same positive connotation, but is used somewhat differently. One of the lexicographer's tasks is to decide how many word senses "charming" has.

In the following sections, I try to define the concept of word sense. In section 1.1, I analyse word senses as being the result of a pragmatic lexicographical process. In section 1.2, I develop the concept of granularity. In section 1.3, I summarize the arguments against and in favour of the concept of word sense.

1.1. Word Sense as Resulting from the Lexicographer's Job

Dictionaries are reference books which catalogue word senses. They are written by lexicographers. What I want to emphasize here is that a lexicographer's job is more pragmatic than one might think.

At first sight, word sense tends to be considered as a universal, unbiased concept. This is not true. A lexicographer has to cope with different requirements:

> In commercial life, the goal is always compromised by practical considerations such as the market that the dictionary is aimed at, its size, its editorial stance, and the speed with which it must be prepared (which may allow, say, twenty minutes per entry)." (Kilgarriff 2006 30)

According to the type of dictionary the lexicographer is writing, he or she decides which word senses to include or reject. So, distinguishing between word senses is not based on an unequivocal concept.

Also, the way lexicographers catalogue word senses is factual. First, they look up a word in a corpus. The programme they use display KWIC[2] lines, i.e. lines containing the word surrounded by a number of words. Then, they gather similar uses into clusters and decide which clusters are worth adding to the dictionary. In other words, they base their judgment on frequency: they choose word senses "that are sufficiently frequent and insufficiently predictable." (Kilgarriff 2006 38) Of course, the results depend very much on the type of corpus they use.

Because of the practicalities of the task, it may not be possible to assess objectively what a word sense is. Nevertheless, it does not mean that there is no such thing as "a word sense" even though the task is context-dependent.

1.2. Granularity

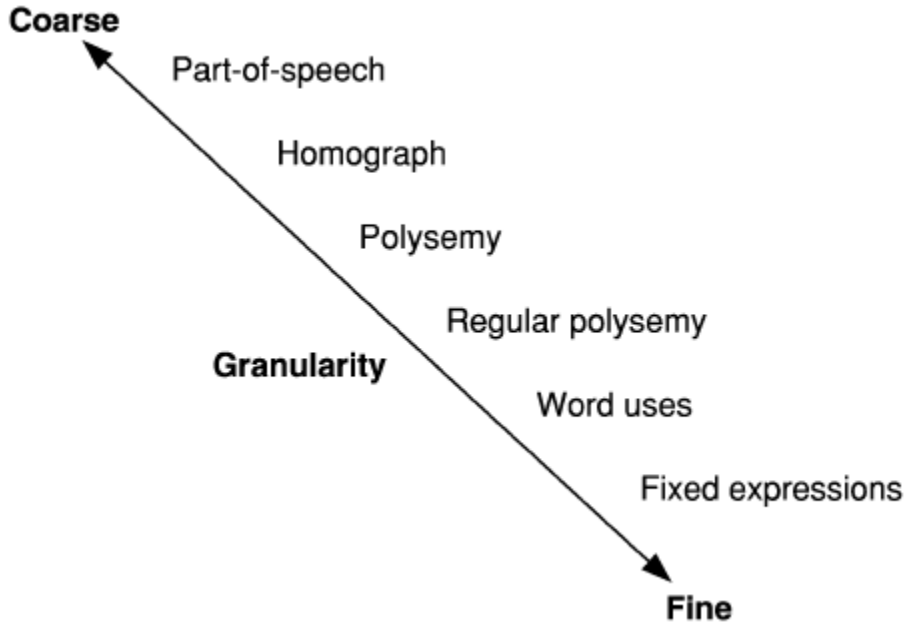
"Word meaning is in principle variable and context sensitive." (Edmonds 608) Most words have overlapping, ambiguous meanings which are not as clear-cut as "mouse" in "computer mouse" or "white mouse". Edmonds illustrates polysemy with the different meanings of the word "bank":

> [*Bank*] has several closely related meanings including:
> - the company or institution
> - the building itself
> - the counter where money is exchanged
> - a money box (piggy bank)
> - the funds in a gambling house
> - the dealer in a gambling house
> - a supply of something held in reserve
> - a place where the supply is held (blood bank). (608-609)

It is far from easy to choose which senses should be included in a dictionary.

[2] key word in context

Some words have clear senses, others do not. In lexical semantics, granularity makes it possible to know to which type a word belongs. Edmonds provides a clear illustration of "the spectrum of distinctions in word meaning in terms of granularity":



(Edmonds 609)

A coarse granularity means that the word senses are too different to be put together. The finer the granularity, the more interrelated the senses.

Part-of-speech ambiguity arises if a word can occur in different parts of speech. For example, "close" can be an adjective ("near", "similar", "stuffy", etc.), a noun ("enclosure", "end") or a verb ("shut").

Words are homographic – and not polysemic – if they are present as distinct words in the speaker's consciousness or if they display an exterior sign (e.g. pronunciation). For example, "row" means either "quarrel"[3] or "queue"[4]. "Port" is another example. Among other things, it means either "wine" or "harbour". In a dictionary, they are considered as two different words and classified in two different entries (e.g. port[1] and port[2]), whereas the senses of a polysemic word are listed in the same entry (e.g. mouse[1-2-3]).

[3] and is pronounced [raʊ]

[4] and is pronounced [rəʊ]

Polysemy is the difficult part. This is exemplified by the different meanings of the word "bank". As Edmonds puts it, "individual senses are often related by a process of extension or modification of meaning – it could be historical, functional, semantic, or metaphorical." (609) For example, "innocent" meant in the first place what it means now, but in a more religious context: blessed women were "innocent". Its meaning broadened to include "simpleton" as the innocent following the doctrine were not critical about it. The adjective underwent a process of pejoration.

Regular polysemy entails predictability, as is the case between object/content (e.g. book) or institution/building (e.g. bank): "I went to the bank" (building) vs. "the bank fired half of its staff members" (institution). In spite of their predictable nature, dictionaries deal with it in their own ways. Some decide to cover it, but even when they do, they are not always consistent.

Word uses and fixed expressions are hard to assess. Is "motor truck" a different kind of truck? Does it deserve its own entry? If it does, does it have to be lexicalized under "motor" or "truck"? If a distinction is made between "truck" and "motor truck", it can only be made on the basis of frequency: "motor truck" appears often enough to be lexicalized even though it remains a kind of truck.

1.3. "I Do Believe in Word Senses"

Several scholars, Kilgarriff included, believe that word senses do not exist. The first argument is the importance of context. In *Lexical Semantics*, Cruse mentions two ways in which context has an influence on the word that is being used: either context *selects* a particular word sense, or it *modulates* the meaning of a word. In "I'm going to the bank because I need cash", only the reading "money bank" is possible whereas in "he pedalled along the bank quite slowly, keeping his eyes skinned for signs of defunct animal life", only the reading "river bank" makes sense. In these examples, context works as a sense filter while in the following sentences, the meaning of "car" is modulated:

- I bought a new car;
- I have to oil the car;
- He's going to pimp out his car.

Different aspects of the car – car as an object, its mechanical parts and its bodywork respectively – are pointed out. If context modulates the meaning of a word, it may mean that meaning cannot

be considered as stable. Therefore, some linguists consider that words have an infinite number of senses as they may appear in any type of context.

Another argument against word senses relates to dictionaries. Dictionaries are authoritative works dealing with word senses. Nevertheless, lexicographers do not write them according to their views on linguistics. They have very practical considerations and guide lines to heed, one of them being to present word senses as distinct as possible: “lexicographers are obliged to describe words as if all words had a discrete, non-overlapping set of senses. It does not follow that they do, nor that lexicographers believe that they do.” (Kilgarriff 1997 100) Furthermore, if word senses are context-dependent, it also seems that they are task-dependent. According to the task that has to be performed, a programme or a human being is likely to need different sets of word senses. Consider again the word “mouse”. In an English to French translation programme, it is not relevant to make a distinction between “device” and “animal” as they both translate to “souris”. But it makes sense to do it in an information retrieval programme insofar as the wrong word sense may cause wrong documents to be retrieved. The task-dependent nature of word senses is also visible at lexicographical level. There exist different sorts of dictionaries, two of them being *monolingual dictionaries* and *learner’s dictionaries*. Monolingual dictionaries help to decode a language while learner’s dictionaries help to encode a language. In other words, a monolingual dictionary tends to be explanatory so that the user can understand the word he or she is interested in whereas a learner’s dictionary is supposed to be more descriptive so that the user can understand and more specifically reproduce the pattern in which the word appears.

Finally, word senses can be construed as mere “[abstractions] from patterns of use.” (Kilgarriff 2006 30) Indeed, lexicographers use corpora from which they infer a word use. Again, this is a very pragmatic perspective that goes against a universal concept of word sense.

In spite of all those arguments, it seems to me that word senses are present at a deeper level than what turns out to be a mere abstraction of use; on the contrary, if abstraction is possible, it seems to prove that the concept of word sense is sound.

It is clear that context may alter the meaning of a word, but it must not be forgotten that people sharing a language often share a similar culture. For that reason, they are not alien to the different contexts in which a particular use might happen. The processes of selection and modulation are so usual that they are not noticed by most native speakers and they present no difficulty to them.

The task-dependent nature of word senses is a more problematic question. On the one hand, having lexical information that suits the task is the best situation, but on the other hand it does not question the concept of word sense. Consider again an English to French translation programme. The point is to say that distinguishing between “mouse-device” and “mouse-animal” is fruitless as they both translate to “souris”. On the other hand, the distinction must be made between “money bank” and “river bank” as they translate respectively to “banque” and “rive”. Such a version can easily be obtained from a dictionary file, simply by merging word senses that produce the same translation. What is important is that it is not possible to create a modified collection of senses without considering all the word senses beforehand. If the task is to translate from English into French, all the word senses have first to be listed and then merged if they give the same translation. The first step is compulsory and backs up the concept of word senses.

2. Word Sense Disambiguation

2.0. Introduction

Word sense disambiguation (or lexical disambiguation) is the process of automatic selection of a word sense in context. If words were not polysemic, a lot of computer applications would be much easier to build. But words are polysemic and a lot of energy is spent trying to solve that problem. In the following sections, I will develop the concepts and the tools that are used to treat lexical disambiguation as well as the fields in which polysemy is an important problem to solve.

2.1. Computational Linguistics (CL) vs. Natural Language Processing (NLP)

CL and NLP are two very close terms, even so that they tend to be interchangeable, but as Wilks mentions in his paper on the history of CL, “NLP [...] is not in itself a program of scientific investigation, which is what CL tends to be.” (761). In other words, NLP needs a task to perform. Typical tasks are machine translation, information retrieval and text summarization. CL is less pragmatic. Among other things, it deals with “a wide range of tasks that are defined only in terms of linguistic theories, and whose outcomes can only be judged by experts.” (Wilks 761) Examples of such tasks are part-of-speech tagging, syntax analysis and WSD. For instance, WSD without an underlining task has no point. It has to be integrated in a programme. Still, WSD is studied as a separate field in CL, the idea being to solve that problem independently. Some

critics argue that this is a mistake as the way lexical disambiguation is handled depends specifically on the task that has to be performed.

2.2. Supervised vs. Unsupervised Systems

There exist a lot of systems dealing with lexical disambiguation and they fall into three categories: supervised, minimally supervised and unsupervised.

Typical supervised systems use manually annotated examples to induce models or rules. The major inconvenient of these methods is that they rely on hand-tagged corpora, a process which demands a lot of time and resources. Yet, supervised systems are generally more efficient than unsupervised systems. These systems can only be statistical: they use the hand-annotated data to induce what the system needs (vectors, rules, models, etc.) in order to be applicable to new information.

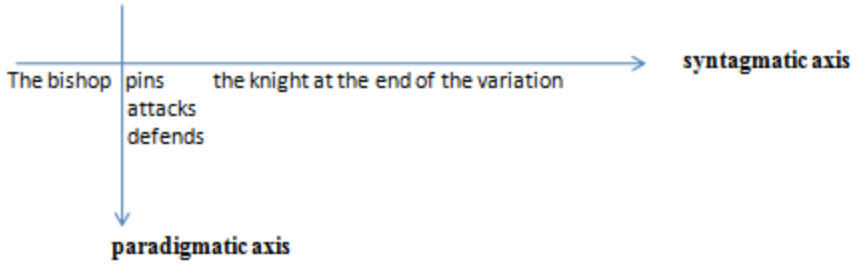
Minimally supervised systems use a very small amount of sense-tagged training examples which create a model that tags other examples. The process is repeated until there is enough data available to start disambiguation. These systems are sometimes labelled “unsupervised”, which can be misleading.

Unsupervised systems are systems like *Lexdis* which do not use any hand-annotated sources. These systems use lexical sources (dictionaries, thesauri, etc.) or rely on statistics.

2.3. Knowledge Sources

2.3.0. Introduction

This section presents different sources that can be used in WSD, generally by unsupervised systems. These sources cover the paradigmatic and syntagmatic axis. Consider the following sentence: “the bishop pins the knight at the end of the variation”. If “pin” is replaced by another verb, it is a change on the paradigmatic axis – “the bishop attacks the knight at the end of the variation” – while if the time adverbial is placed elsewhere in the sentence, it is a change on the syntagmatic axis – “at the end of the variation the bishop pins the knight”:



It is important that lexical sources should cover the two axes. Indeed, the aim of a WSD tool is to know which word sense is intended. The relationship between the word to disambiguate and context (e.g. another word) may be of paradigmatic or syntagmatic order. For instance, collocations are only present at syntagmatic level whereas the relationship hypernym-hyponym is more visible at paradigmatic level.

Chapter 4 describes the lexical databases that are used by *Lexdis*. A lot of them are part of the knowledge sources described below. While I describe in chapter 4 how they are structured and what they contain, in the following sections I aim to give a general overview of all the possible lexical resources and their relevance to the question of WSD.

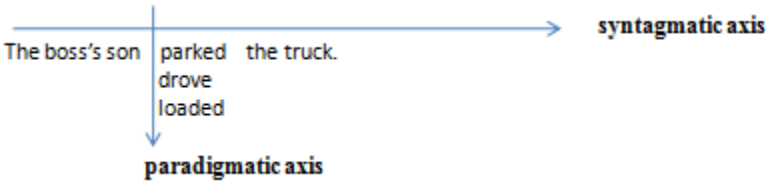
2.3.1. Dictionaries

Dictionaries are the knowledge source about word senses *par excellence* insofar as their main aim is to catalogue word senses. All the information they provide can be useful: definitions, example sentences, collocations, synonyms, etc. There exist many types of dictionaries and some suit WSD better than others. For instance, learner's dictionaries provide somewhat more relevant lexical information than other monolingual dictionaries as they help to encode a language while other dictionaries are more clearly geared towards decoding a language. Bilingual dictionaries are also very helpful. Only the language in which lexical disambiguation is taking place is used (in this case, English). The ultimate purpose of bilingual dictionaries is similar to that of learner's dictionaries: they aim at indicating the pattern in which a word occurs – pattern being here collocations, expressions, phrases, idioms, uses, etc. – that they translate into the target language.

That being said, the other types of dictionaries contain valuable lexical information. Example sentences, but also definitions from monolingual dictionaries are very helpful resources. The idea remains to use as many quality lexical sources as possible.

2.3.2. Collocations

Collocations are words that “often occur together” (Collins Cobuild). So collocations are only present at syntagmatic level. For instance, according to the OCD (Oxford Collocations Dictionary for Student of English), “truck” is often associated with the verbs “drive, park, load/unload”. But of course, these verbs can form a paradigm:



Any other verb can be added to it (e.g. “to paint”), but in that case, the paradigm does not consist only of collocations anymore. More information on collocations is to be found in chapter 3.3.

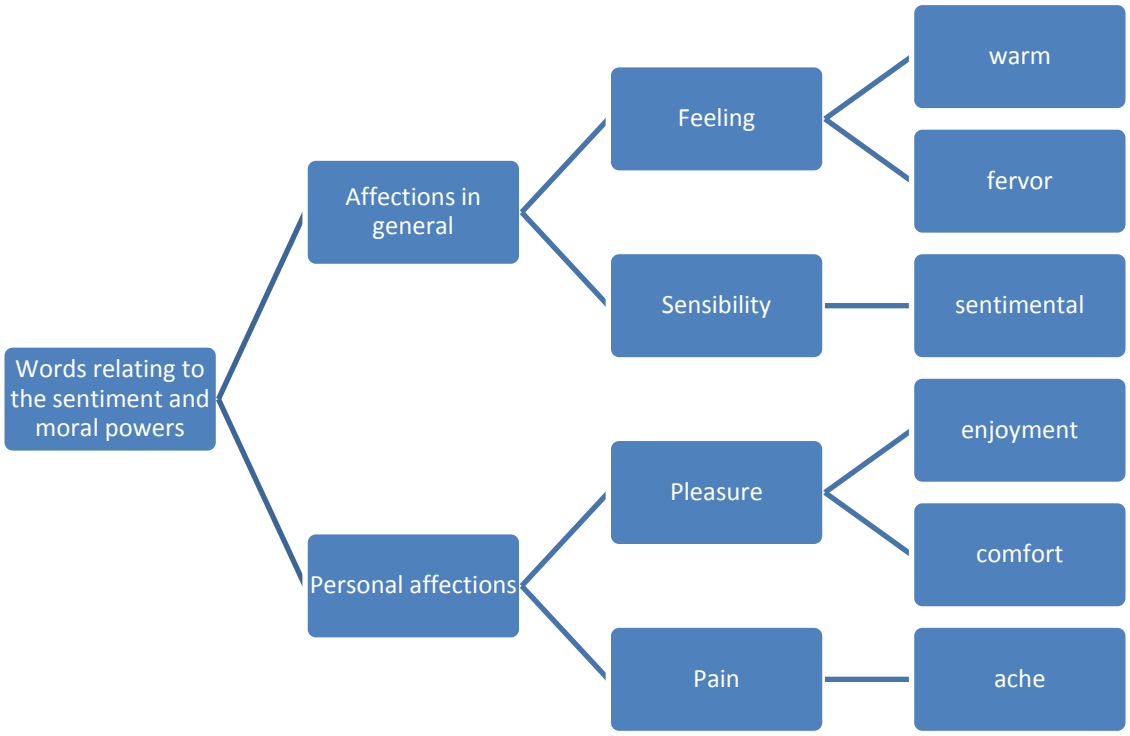
2.3.3. WordNet

WordNet is a well-known lexical resource developed by a team of researchers working at Princeton University. All the relationships revolve around synsets which are a group of words that are synonyms in a particular set. The semantic relationships include: gloss (definition), hypernymy-hyponymy, entailment, meronymy, similarity (only for verbs), antonymy and synonymy. WordNet covers the syntagmatic axis, but a lot more the paradigmatic axis than dictionaries for instance. Another important feature is that it is freely available. See chapter 4.7 for an illustration of a synset.

Two things should be kept in mind when using WordNet for lexical disambiguation. First of all, it contains lots of specific scientific words which most general dictionaries do not cover. But more importantly, the average number of word senses per word is rather high. In other words, the level of granularity is often too fine for applications that want to remain as general as possible. For example, the third version of WordNet has more than 8 word senses for “to represent”, while monolingual and bilingual dictionaries have about 4.

2.3.4. Thesauri

A thesaurus "lists words and groups of synonyms and related concepts." (COED[5]) Again, it focuses on the paradigmatic axis. A famous and freely available thesaurus is Roget's Thesaurus. It uses a system of classes, categories and sub-categories. Here is a small sample of a class and its sub-class:



2.3.5. Pragmatics

In some cases, knowledge of the world is necessary to disambiguate a word. Consider "the box is in the pen". In this case, "pen" can only be an enclosed space as a box would never fit in a pen (i.e. the writing instrument). This may be the most difficult field to cover computationally.

2.3.6. Financial Considerations

Quality dictionaries are nothing out of the ordinary. Think about Le Grand Robert & Collins 2009, the OED, the Oxford or Cambridge Advanced Learner's Dictionary. If their database were to be available in an open source format, they could be used in the field of WSD. Obviously, the companies that sell the dictionaries will never distribute their work freely. The paradox is that a

[5] Concise Oxford English Dictionary

lot of quality machine readable resources are now available, but the financial market makes it almost impossible to use them in the field of WSD or in a task that requires lexical disambiguation.

2.4. Tasks

2.4.0. Introduction

In the following sections, I am going to present the different areas in which WSD could improve the results of a particular task. The purpose of section is not to present a full coverage of these tasks in relation with WSD. I simply aim at introducing several fields in which WSD may improve the results of an application.

2.4.1. Information Retrieval

In monolingual IR, WSD seems to actually give worst results according to Sanderson (1994) and Voorhees (1999). This comes from the retrieval technique that is generally used. The idea is that the user enters a query of any number of words to get documents that match the query. If the query is too short (i.e. a couple a words), there is not enough clue to perform disambiguation efficiently. If the query is longer, disambiguation is often not necessary as the algorithm uses a “bag-of-words” technique. It means that a document is seen as an unordered collection of words to which the query is matched. In this case, the words that accompany the word to disambiguate are enough to provide disambiguation. Say you google something like “explanation financial crisis banks bankrupt”, it is very unlikely that you get documents about “river banks” because the words “financial”, “crisis” and “bankrupt” already disambiguate “banks”. This recalls the selection process of context mentioned by Cruse (cf. 1.3).

In cross-language IR, WSD is more efficient. The process amounts to translating the query into another language to get more documents. In this case, if the system translates “explanation financial crisis banks bankrupt” by “explication financier crise rive faillite”, that is not going to do. WSD is useful because the contextual disambiguation does not take place.

Note that IR systems all use “stemming”, i.e. a system that links words to their roots. For instance, “connect”, “connects” and “connection” are all triggered in a query that contains the word “connections”.

2.4.2. Question Answering

QA resembles IR. Like IR, it retrieves documents, but its purpose is to get the exact answer to the user's query. With the query "who invented the computer", the point is not to get documents about the history and development of the computer, but the name of the inventor. Because QA systems need to deal with semantics, it is believed that word sense distinction can improve the quality of the answers. For an example of a QA system, have a look at *Wolfram Alpha* website[6].

2.4.3. Machine Translation

Any sentence to translate from English into French that contains a polysemic word is potentially problematic for an MT system. For example, in "the car ran out of gas", "gas" should translate to "essence" and not to "gaz", and "car" to "voiture" and not "cabine d'ascenseur".

[6] http://www27.wolframalpha.com/. For instance, the query "distance earth moon" directly gives 365 377 km. The query "what is the meaning of life" obviously gives 42.

Chapter III: How does *Lexdis* work?

0. Introduction

When Edmonds asserts that "many words are open to different semantic interpretations depending on the context" (607), he is stating the obvious. It is a well-known fact that most words are polysemic. Nonetheless, Edmonds points out that context is the key to interpretation, or in other words to disambiguation, and any other word can be perceived as context. *Lexdis* is based on the assumption that with lexical information, the scope of polysemy of the first word narrows the scope of polysemy of the second word and vice versa. Consider the pair "break" (verb) and "mouse" (noun) from the sentence "I broke a mouse". The verb "break" has at least the following meanings: "I broke that glass", "a scandal broke yesterday", "a storm is going to break" and "I'll break for lunch". The noun "mouse" refers either to the animal, to a computer device or to a very timid person. The idea is to "computationally determine which sense of a word is activated by the use of the word in a particular context" (Edmonds 607). Note that a possible plural of "mouse" as a computer device is "mouses" instead of "mice", but since this lexical form is attested only in recent dictionaries, it will not be used to disambiguate "mouse". So *Lexdis* is supposed to produce a higher number with the word senses "break a glass" and "computer mouse" than with other combinations.

Even though a lot of examples are taken from the field of word sense disambiguation, it must be noted that *Lexdis* is not a fully-fledged WSD programme. *Lexdis* only supports lexical queries, for instance "break" as a verb with "mouse" as a noun[7]. It is not able to analyse a text or to parse a sentence. Consider Bar-Hillel's famous sentence "the box was in the pen" in which the difficulty consists in finding the correct word sense of pen, i.e. an enclosed space and certainly not a writing instrument. The only query that *Lexdis* supports is "box and pen as nouns", but then, the original sentence might have been "I used a pen to write on a box" or even "the pen was in the box". In other words, *Lexdis* is an enabling application. It is not a complete programme which fully covers lexical disambiguation; it merely computes lexical proximity.

In the following sections, I describe how *Lexdis* works. Section 1 deals with Lesk's famous algorithm and its implementation in *Lexdis*. Section 2 comments on the type of knowledge

[7] The corresponding *Lexdis* query is [break,v,mouse,n].

source that *Lexdis* uses to compute lexical proximity. Section 3 defines collocations and emphasizes their importance as a knowledge source in computational linguistics. In section 4, I describe *Lexdis*'s two main query modes. In section 5, I describe how *Lexdis* is used to merge homologous definitions coming from different dictionaries or lexical resources. Finally, in section 6, I describe the algorithm that was implemented to compute lexical distance between three words.

1. Lesk and *Lexdis*'s Algorithm

In his well-known paper "How to tell a pine cone from an ice cream cone", Michael Lesk introduces a simple algorithm that "decide[s] automatically which sense of a word is intended" (24). The example he first develops deals with "pine cone" and "ice cream cone". The programme starts by looking at the definitions it can find in the Oxford Advanced Learner's Dictionary of Current English. It finds that "pine" has two main senses, i.e. "kind of evergreen tree with needle-shaped leaves..." and "waste away through sorrow or illness..."; and that "cone" has three: "solid body which narrows to a point...", " something of this shape whether solid or hollow..." and "fruit of certain evergreen trees...". The idea that underpins Lesk's algorithm is simple: to count the overlapping words between definitions. It appears that "evergreen" and "tree" are common to the two intended senses. Lesk also tries his programme on the famous "time flies like an arrow; fruit flies like a banana" and it gives good results. The only word that is incorrectly disambiguated is "arrow" as a sign and not as a stick.

Lexdis uses a similar algorithm. It uses lexical databases (cf. next section) as knowledge sources. When a query is launched, it fetches all the lexical information available in the main database and counts overlapping words in definitions and example sentences. *Lexdis* also uses other databases (cf. next section), but the idea remains to see whether the elements from the query have something in common. For instance, *Lexdis* checks whether the two words share the same guide word or the same category (or subcategory) in Roget's Thesaurus. Nevertheless, it must be noted that Lesk implemented his algorithm in a programme that had several options that do not have any equivalent in *Lexdis*: it accepts whole sentences and has a customizable window that goes from 4 to 10 words. *Lexdis* accepts queries of the following type: first word, part of speech, second word, part of speech, mode. So the syntactic ambivalence in "time flies like an

arrow; fruit flies like a banana" that Lesk's programme is able to resolve cannot be solved by *Lexdis*.

Lesk's algorithm as well as *Lexdis*'s depends on the lexical information they are fed with. When Lesk tested his programme in 1986, machine readable dictionaries (MRD) were not as developed as they are now. Also, certain types of lexical information started to be included in dictionaries later. For example, collocations (cf. section 3) are more homogenously recognized as such and included in example sentences from the end of 70s. *Lexdis* takes advantage of a wide range of different lexical information that was unavailable in the late 80s. Lesk also wonders how to count the overlapping words, but since *Lexdis* uses different knowledge sources, it is possible to favour a particular type of information and assign it a greater weight.

Lesk also introduces a system in his programme whose aim is to weight the final result according to the length of the entry in the dictionary since the longer an entry, the more chances there are that the programme finds overlaps. In *Lexdis*, the weighting system consists of a file containing the lexical weight of each lemma that *Lexdis* uses to reduce a number that may not stand for the actual proximity between two words.

For a detailed account of the evaluation of *Lexdis* in relationship with the different lexical weights, see chapter 6.

2. Knowledge Source: Lexical Databases

Lexdis uses lexical databases to compute lexical proximity. It does not rely on statistics. There exist several programmes which give an interesting treatment of meaning in computational linguistics, in the field of word sense disambiguation or text summarization for example, but most of them use statistics to achieve their ends. These programmes feed on large quantity of information to refine their algorithms. *Lexdis*'s algorithms are not altered by the lexical databases; they are defined heuristically and empirically.

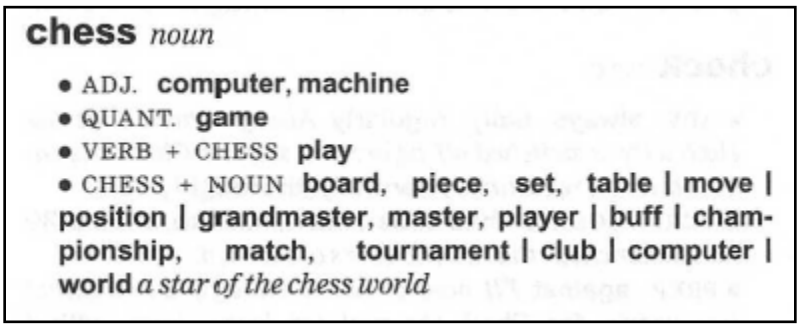
The lexical databases used by *Lexdis* are the following: the two bilingual dictionaries Robert-Collins and Oxford-Hachette (in the direction English to French); the monolingual dictionaries CIDE, LDOCE, Collins Cobuild; Roget's Thesaurus and WordNet. From the bilingual dictionaries, *Lexdis* uses definitions, example sentences, collocations and indicators. From the monolingual dictionaries, *Lexdis* uses the definitions and the example sentences. From Roget's

Thesaurus, *Lexdis* uses its hierarchical classification of lemmas. Finally, *Lexdis* uses WordNet's glosses and synsets as well as the relationships between synsets.

As mentioned above, *Lexdis* counts the overlapping words from databases. The number of shared words is weighted if necessary. It is important that word senses should meet only when it is significant and relevant. To make sure to avoid noise, all the databases have been filtered by a home-made stop list, which is a list containing very frequent words such as determiners (e.g. "the", "a") or words often used in definitions such as "especially". Consider the following definition of bishop: "in the game of chess a bishop is a piece sometimes in the shape of a bishops hat that can only move diagonally along squares of the same colour". (CIDE) After being filtered, the definition contains only relevant words: "game, chess, shape, bishops, hat, move, diagonally, squares, colour". Note that the frequent word "sometimes" was deleted. The importance of a good stop list must not be underestimated as the quality of the database partly depends on it.

3. Collocations

Although collocations have been considered to be of pedagogical interest from 1930s, JR Firth is the first linguist to theorize them 20 years later. He called them "an abstraction at syntagmatic level" (1957 196). Already, the definition bears the idea that meaning is not merely the sum of the parts that make up the sentence. More recently, collocations have been defined as "any statistically co-occurrence [of words]" (Sag et al. 8), which implies that the study of collocations is corpus-based. The OCD[8] points out that collocations are part of everyday speech: "collocation is the way words combine in a language to produce natural-sounding speech and writing" (Lea, vii). Here is an entry example from the OCD:



chess *noun*

- ADJ. **computer, machine**
- QUANT. **game**
- VERB + CHESS **play**
- CHESS + NOUN **board, piece, set, table | move | position | grandmaster, master, player | buff | championship, match, tournament | club | computer | world** *a star of the chess world*

[8] Oxford Collocations Dictionary for Student of English

The OCD provides very useful lexical information, even for more specific lemmas. Even recent learner's dictionaries such as the OALD[9] or the CALD[10] fail to provide such detailed information about the collocations of a lemma.

There exist different types of collocations, ranging from syntactically free to almost or completely frozen. For instance, a collocation like "to see a film" is syntactically very open: "to see" can be conjugated in any tense, its polarity is not determined by the collocation, "film" can be in the singular or plural and can take an adjective. Already, a "motor truck" is syntactically less open to alterations: nothing can split up "motor truck"; "motor" is invariable and cannot be replaced by a near-synonym like "engine". "Not to see the wood for the trees" is almost frozen: "to see" can be conjugated in any tense, but its polarity must be negative and its object is invariable. The more frozen the collocation, the more they tend to be considered as idioms: "[an idiom is] a group of words in a fixed order that have a particular meaning that is different from the meanings of each word understood on its own" (CALD). For instance, the OCD, whose editor decided not to include idioms, does not feature "not to see the wood for the trees" meaning "to be unable to get a general understanding of a situation because you are too worried about the details" (CALD), but includes "to drive a hard bargain" meaning "to expect a lot in exchange for what you pay or do" (ibid.) because the latter still has something to do with "bargain".

Collocations are a very valuable knowledge source in computational linguistics. Actually, collocations are statistical information turned into lexical information. It is much more reliable than corpus-based information inasmuch as it is controlled by lexicographers. What is more, collocations can be used to "distinguish [word] senses" (Krishnamurthy 597). Indeed, The OCD distinguishes two word senses for the noun "mouse", i.e. "animal" and "for a computer". For instance, the collocation "mouse trap" will always apply to the former word sense and "mouse click" to the latter. Also, when Michael Lesk wants to make clear that "cone" has at least two distinct senses, he uses collocations: "pine cone" and "ice cream cone".

The way *Lexdis* uses collocations is explained in chapter 4.1.

---

[9] Oxford Advanced Learner's Dictionary
[10] Cambridge Advanced Learner's Dictionary

4. Global and Local Mode

Two query modes have been implemented in *Lexdis*, i.e. a local mode and a global mode. The local mode only deals with word senses. In a typical two-word query in local mode, *Lexdis* fetches all the words senses of the two words and computes lexical proximity between all the pairs. It is therefore possible to see which pair has the highest number or to see the discrepancy between the different pairs. In local mode, *Lexdis* works as a WSD tool.

Even though it has not been implemented that way, the result of a two-word query in global mode amounts to the sum of the results in local mode. In other words, this mode makes no distinction between word senses. For example, this mode can be used with a parser. Consider that the parser contains the following information about the collocations of the phrasal verb “to go through”. Each collocation set distinguishes a new word sense to which an example sentence is added:

- consume: stock, store, beer, food, fortune; e.g.: it didn't take Albert very long to go through his inheritance;
- search: room, pocket, paper; e.g.: it was obvious that the room had been gone through by an intruder;
- perform: marriage, initiation, matriculation, ceremony; e.g.: they finally went through the marriage ceremony for the sake of their children;
- rehearse: fact, argument, scene, text; e.g.: they went through the details of the plan over and over again;
- be published: printing, edition; e.g.: his book went through ten editions in a year;
- experience: operation, pain, ordeal, fire; e.g.; he would have gone through fire for the girl he loved.

The parser is able to analyze the sentence “I think that they went through five bottles”, but the user would like to know which word sense of “go through” is intended. All he or she has to do is compute lexical proximity between the object of the sentence (i.e. bottle) and each collocation of each word sense, in global mode. The collocation set that features the highest mean is supposedly the correct word sense. In this case, *Lexdis* should conclude that “consume” is the best match, which it does. In this example, *Lexdis* is used in global mode because we are not interested in the word senses of the collocation and the object, we want to know how strongly they relate. This example is commented on in chapter 6.3.2.

5. Merging the Entries in the Main Database

Lexdis uses definitions and examples from various sources. On the one hand, this is a good thing because it provides more information to *Lexdis* which can compute lexical proximity more accurately. On the other hand, the structure of the sources is not optimal in local mode. Consider the noun “mouse”. It is more than likely that all the sources distinguish at least two word senses, i.e. computer device and animal. If you launch a query with “mouse” and another word in local mode, what you get is a result for each word sense from each source. Of course, it would be more productive to have only one result for each word sense, whatever their origin may be. This can be achieved only by merging the entries of the main database wherever possible.

The merges are performed by *Lexdis*. The first merge query is: word, part of speech, same word, same part of speech, merge mode, local mode. The candidates for merging are the word whose lexical weight is greater than 1. This minimal threshold makes sure that there is enough information about the word in the database. *Lexdis* computes the lexical proximity of each pair and merges the one with the highest result. This process is applied to all the words in the database. The merge mode keeps track of the origin of the merged entries and another programme removes the source entries that were merged.

Merging the databases this way is supposedly safe. Indeed, it is very unlikely that two wrong word senses are merged. Consider again the word “mouse”. A wrong merge would consist of “mouse” as a computer device from one dictionary with “mouse” as an animal from another dictionary. This occurs only if the main database has one occurrence of the word sense “computer mouse” and several occurrences of the word sense “white mouse”. Even then, the wrong pair would have to produce a higher number than the other good pairs (i.e. “white mouse” with “white mouse”). Another possibility is that there is only one attested word sense of “computer mouse” and one of “white mouse”. Considering that the main database is made up of three learner’s dictionaries and WordNet’s glosses and synsets (cf. chapter 5.8), it is more than unlikely that these worst case scenarios occur.

A second merge is performed, this time aiming at lighter entries. The candidates are only the words with a lexical weight of 1, but a minimal threshold is applied on the result (10 in this case) to make sure of the quality of the merge.

The third merge does not include any minimal lexical weight, but forces a greater lexical proximity (of 16), once again to only allow quality merging. This merge is repeated until the number of merged entries is not high enough to justify going on with the process.

Not only does the resulting database contain better entries, but the file is lighter (from 121.173.585 KB to 99.681.061 KB), which means that it loads somewhat faster.

6. Triplets

Lexdis also supports three-word queries, but the algorithm is based on two-word queries. First of all, a new mode that only displays the three best possible matches (or less if there are less than three) is implemented. Consider the sentence "he was wearing a gold tie and a red watch" (6a). The distances that would be interesting to compute are between "wear" as a verb and "tie" and "watch" as nouns. The three-word query takes the first word as an argument bearer, and two words as arguments. Most of the time, the argument bearer is a verb and the two other arguments are nouns. *Lexdis* computes lexical proximity this way: argument bearer – first argument; argument bearer – second argument; first argument – second argument. Then, it checks whether a word sense is common to two of the queries. If *Lexdis* is able to produce a result for each three words, then these word senses are the result of the query.

Consider sentence 6a that gives the following query: "wear" as a verb, "tie" as a noun, "watch" as a noun. A schematic positive result may be:

wear – tie	wear – watch	tie watch
wear ws1 and tie ws1	wear ws1 and watch ws1	tie ws1 – watch ws1
wear ws2 and tie ws2	wear ws4 and watch ws1	tie ws1 – watch ws2
wear ws3 and tie ws3	wear ws5 and watch ws1	tie ws1 – watch ws3

The definitions in relationship with the word senses are:

- wear ws1: to have clothing or jewellery on your body;
- wear ws2: to produce something such as a hole or loss of material by continuous use rubbing or movement;
- wear ws3: if you wear a particular expression your face shows the emotions that you are feeling a fairly literary use;

- wear ws4: put clothing on one's body; "what should I wear today?"; "he put on his best suit for the wedding"; "the princess donned a long blue dress"; "the queen assumed the stately robes"; "he got into his jeans";
- wear ws5: go to pieces; "the lawn mower finally broke"; "the gears wore out"; "the old chair finally fell apart completely";
- tie ws1: neckwear consisting of a long narrow piece of material worn (mostly by men) under a collar and tied in knot at the front; "he stood in front of the mirror tightening his necktie"; "he wore a vest and tie", a tie is a long narrow piece of cloth that is worn round the neck under a shirt collar and tied in a knot at the front ties are worn mainly by men see also bow tie old school tie, also esp ame necktie a band of cloth worn round the neck usu inside a shirt collar and tied in a knot at the front, a tie also esp am necktie is a long thin piece of material that is worn under a shirt collar esp by men and tied in a knot at the front;
- tie ws2: a long narrow piece of cloth plastic wire etc that is used to attach one thing to another or to close or fasten something such as a bag or a piece of clothing;
- tie ws3: a tie is also a connection relationship or feeling that links a person with another person a place an organization etc;
- watch ws1: a small clock which is worn on a strap around the wrist or sometimes connected to a piece of clothing by a chain, a watch is a small clock which you wear on a strap on your wrist or on a chain;
- watch ws2: often in comb a small clock to be worn or carried;
- watch ws3: also night watch a form of police force doing duty in towns at night in former times made up of citizens serving in turn or else of paid men with no special training.

The result is positive because there are word senses which are common to two queries and because *Lexdis* selected the right word senses. The definition of “tie ws1” is rather long as it is the result of a merge.

Note that the triplet mode is tested in chapter 6.4.

Chapter IV: Description of the Databases

0. Introduction

A comprehensive description of the databases is essential to a good understanding of *Lexdis* insofar as it could not work without them. *Lexdis*'s aim is to relate words, but that would be impossible without preestablished information.

This chapter describes how the databases are structured, the information they contain as well as a brief explanation of the algorithm that uses the databases in question. A full explanation of the weighting algorithm is to be found in chapter 6.

1. coll.pl

This file contains the collocate lists derived from two bilingual dictionaries the Robert-Collins and the Oxford-Hachette. Collocates are words that "often occur together" (Collins Cobuild). For example, the collocates associated with the lemma "mean" are:

```
1  coll(lemma('mean'),pos(adj),coll(['appearance','existence'])).
2  coll(lemma('mean'),pos(adj),coll(['attitude','nature'])).
3  coll(lemma('mean'),pos(adj),coll(['behaviour','action'])).
4  coll(lemma('mean'),pos(adj),coll(['birth'])).
5  coll(lemma('mean'),pos(adj),coll(['city'])).
6  coll(lemma('mean'),pos(adj),coll(['distance','temperature','price','weight'])).
7  coll(lemma('mean'),pos(adj),coll(['dwelling'])).
8  coll(lemma('mean'),pos(adj),coll(['examiner'])).
9  coll(lemma('mean'),pos(adj),coll(['exponent','shot'])).
10 coll(lemma('mean'),pos(adj),coll(['horse','dog','animal','expression'])).
11 coll(lemma('mean'),pos(adj),coll(['origin'])).
12 coll(lemma('mean'),pos(adj),coll(['street'])).
13 coll(lemma('mean'),pos(adj),coll(['trick'])).
14 coll(lemma('mean'),pos(v),coll(['budget','tax','cuts','cut'])).
15 coll(lemma('mean'),pos(v),coll(['gift'])).
16 coll(lemma('mean'),pos(v),coll(['remark'])).
17 coll(lemma('mean'),pos(v),coll(['sign'])).
18 coll(lemma('mean'),pos(v),coll(['strike','law','shortages','changes','shortage','change'])).
19 coll(lemma('mean'),pos(v),coll(['word','symbol','phrase'])).
```

The pivot is the collocate bearer, i.e. the word "mean". It is quoted, i.e. inserted between quotes, to avoid any compatibility problem, just as the collocates are. The complete file is alphabetically sorted. My sample numbers nineteen lines. Each of them bears collocates for a different word sense of "mean". First of all, it is obvious that the adjective "mean" should display different collocates from the verb "to mean". This is one reason why the part of speech ("pos") was kept. But even the adjective "mean" is polysemic: "mean" as "average" or as "unkind".

To allow all the words to relate to each other in the main programme, the lemmatized form – i.e. the singular form – of the plural nouns is added: "shortage" and "change" (line 18) are produced from "shortages" and "changes". The downside of this is that some words may not be depluralized without a change in meaning. Some words can easily be added to a stop list so that their singular form is not added – as for instance the singular noun "means" – but in some cases, this is not feasible. Take for instance the word "spectacles". It cannot be depluralized when meaning "a pair of spectacles", but only when meaning "a show". It would be inappropriate to try and identify each plural form: the data is generated by a computer programme and sorting the plural nouns on an individual basis would bias the result at best, produce mistakes at worst.

The idea is to posit that two words are semantically linked if one or several words are common to the two collocate lists. The strength of the link is proportional to the number of collocate words they share: the more words, the stronger. Let's say we want to relate the adjective "mean" to the adjective "average". While the collocates of "mean" are to be found in the previous array, the collocates of "average" are:

```
coll(lemma('average'),pos(adj),coll(['amount','cost','earnings','rate','earning'])).
coll(lemma('average'),pos(v),coll(['distance','quantity','time'])).
```

The two words share one collocate: "distance". The programme heeds this piece of information to increase a number associated with a linking strength.

2. mt.pl

Just as *coll.pl*, this file was produced from the Robert-Collins and the Oxford-Hachette. What differs is the organisation of the data:

```
mt(circumstances,[[colour,1],[condition,4],[conditions,3],[death,1],[decision,1],[epoch,2],[event,9],
                  [events,3],[face,1],[fact,1],[fate,1],[feature,1],[goal,2],[government,1],[idea,1],
                  [ideas,1],[importance,1],[incident,1],[information,1],[leadership,1],[lifestyle,1],
                  [meeting,1],[motive,1],[need,2],[needs,2],[object,1],[occasion,1],[occurrence,1],
                  [opinion,1],[person,2],[plan,2],[plans,2],[price,1],[publicity,1],[rate,1],[reaction,1],
                  [reason,1],[requirement,1],[requirements,1],[residence,2],[route,1],[shot,2],
                  [situation,5],[size,1],[skill,1],[speed,1],[state,2],[technique,2],[techniques,2],
                  [timetable,1],[vehicle,1],[weather,4],[wind,1],[work,1],[year,2]]
    ).
```

The pivot is the word "circumstances". All the other words happen to be present a number of times in a collocate list to which "circumstances" also belongs: "circumstances" and "event" co-occur in nine different collocate lists. This data was actually produced from *coll.pl*:

```
coll(lemma('concatenation'),pos(n),coll(['ideas','events','idea','event','circumstances','circumstance'])).
```

For instance, "concatenation" from *coll.pl* is one of the nine meeting points of "circumstances" and "event".

The hypothesis, first put forward by Montemagni et al. (1996), is to say that if two words are part of the same collocate list, they are semantically linked. The more times they occur in the same collocate list, the stronger the link.

To avoid duplicates, the first word in alphabetical order is the pivot of the list. For instance, "event" and "circumstances" co-occur nine times and each time they do, "circumstances" is always taken as pivot since it comes first in alphabetical order. That way, the information is not repeated when "event" is the pivot. Lists become shorter by the end of the file as the file is alphabetically sorted.

3. indic.pl

Like the previous files, it comes from the two bilingual dictionaries the Robert-Collins and the Oxford-Hachette. It contains indicator lists associated to a pivot:

```
ind(lemma('abroad'),pos(adv),indic(['doors','outside'])).
ind(lemma('abroad'),pos(adv),indic(['far','wide','directions'])).
ind(lemma('abroad'),pos(adv),indic(['far','wide'])).
ind(lemma('abroad'),pos(adv),indic(['foreign','land'])).
```

Indicators are special fields that help to choose a particular translation in a bilingual dictionary. Since the programme is designed only for English, only the direction English to French was explored. To give a better idea of what indicators are, here is part of the entry "abroad" in a more recent version of The Robert-Collins:

The indicator is "in foreign country". Obviously, it would not make sense to keep words such as "in", "all", "of" in the lists. That is why all these high frequency words were added to the stop list.

In the same way as for the previous files, the programme explores the indicator lists of the two words to check whether they share indicators:

```
ind(lemma('advance'),pos(v),indic(['chess'])).
ind(lemma('bishop'),pos(n),indic(['chess'])).
```

The hypothesis is to claim that "advance" and "bishop" are semantically linked because they share "chess" in their indicator field.

4. roget.pl

This file contains a modified version of Roget's Thesaurus whose most recent version dates from the beginning of the twentieth century:

```
r('matutinal',[['adj','125','5','1']]).
```

The first part of the entry is the lemma. There follows a list that contains all the categories to which the lemma belongs. For instance, the lemma "matutinal" as adjective appears only in one category, the 125th one, in the sub-category 5 and in the sub-sub-category 1. The structure becomes clearer when compared with the source (Roget's Thesaurus):

1	#125. Morning. [Noon.] -- N. morning, morn, forenoon, a.m., prime, dawn, daybreak; dayspring[obs3], foreday[obs3], sunup;
2	peep of day, break of day; aurora; first blush of the morning, first flush of the morning, prime of the morning; twilight, crepuscule, sunrise;
3	cockcrow, cockcrowing[obs3]; the small hours, the wee hours of the morning. spring; vernal equinox, first point of Aries.
4	noon; midday, noonday; noontide, meridian, prime; nooning, noontime. summer, midsummer.
5	Adj. matin, matutinal[obs3]; vernal.
6	Adv. at sunrise &c. n.; with the sun, with the lark, "when the morning dawns".
7	Phr. "at shut of evening flowers" [Paradise Lost]; entre chien et loup[Fr]; "flames in the forehead of the morning sky" [Milton]; "the breezy call of incense-breathing morn" [Gray].

Each paragraphs starts with a lemma. Each sub-category is separated by a line break, to which I added an extra one so that it stands out. The sub-sub-category separator is the semicolon.

Lexdis uses only the numbers associated with the lemma to relate words:

```
r('maturity',[['n','124','1','5'],['n','673','7','3']]).
r('mellowness',[['n','673','7','2']]).
```

The two words are related because they share at least one category (673) and in this case one sub-category (7). There is only one better case: the sharing of the three categories. In other words, when trying to relate two words, the programme analyzes their category information.

5. envir.pl

Envir.pl was also produced from the two bilingual dictionaries the Robert-Collins and the Oxford-Hachette, this time from the example phrases and sentences. The pivot is not a lemma strictly speaking, but an extended version of it:

```
e(hdwd('all over'),envir(['spots','arms','trembling'])).
```

Actually, "all over" is to be found under the lemma "all". The entry consists of a headword and a list of words. For instance, the word "trembling" comes from the example "he was trembling all over". As said for *indic.pl*, it would not be semantically relevant to retain frequent words such as "he", "all" or even "specially", a word that is very often used in definitions and examples. This would only create noise inasmuch as words that would normally not be related would meet via these frequent words.

As usual, *Lexdis* takes the lists from the words the user asked to relate and checks whether they have one or more words in common.

6. pesi.pl

All the previous files were produced from databases and contain diverse types of information about lemmas. It is a fact that some words get a more extended lexicographical treatment: any dictionary provides more information on "great" than on "matutinal". This file aims at reducing the gap between "heavy" and "light" commented words:

```
w(principles, n, 50).
w(prink, v, 81).
w(print, adj, 58).
w(print, n, 634).
w(print, v, 413).
```

It consists of a lemma associated with a number. The higher the number, the more information there is in the databases about the lemma. It was created by running the following query into *Lexdis*: [Word,Pos,Word,Pos,m:g]. *Lexdis* computed lexical proximity between a word and the very same word. The more information available in the database on a word, the higher its weight.

The use of the weighting file is not compulsory. On the one hand, if a large amount of information is available on a word, it means the word is frequent and the number representing

the lexical proximity is supposed to be high. On the other hand, it may be more difficult to compare that number with another number to get an idea of their relative lexical proximity. Take the following pairs, "work – car" and "organize – party":

```
w(car,n,311).
w(work,v,1775).
w(party,n,69).
w(organize,v,360).
```

If we want to compare the discrepancy between the pair “work” and “car” and the pair “organize” and “party”, it may be wise to use the weighting system. “To work” is a very general verb and accordingly the result is going to be high. Without a weighting system, the gap between these two numbers might be too broad to represent a realistic relative semantic proximity.

7. semdic.pl

This file is the biggest file. It contains dictionary definitions and examples from the Cambridge International Dictionary of English (CIDE), the Collins Cobuild, the Longman Dictionary of Contemporary English (LDOCE), and WordNet synsets and synset glosses. The first three sources are learner’s dictionaries; WordNet is different: “WordNet® is a large lexical database of English [...]. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated [...].” (WordNet) Consider synset 202627934. The definition and the examples to be found in a worked version of the WordNet file are:

```
g(202627934,
   deflex(['require','useful','proper']),
   exlex(['nerve','success','requires','hard','work','job','asks','patience','skill','position','demands','personal',
          'sacrifice','dinner','calls','spectacular','dessert','intervention','postulate','patients','consent']),
   def('require as useful, just, or proper; "it takes nerve to do what she did";
       "success usually requires hard work"; "this job asks a lot of patience and skill"; "this position
       demands a lot of personal sacrifice"; "this dinner calls for a spectacular dessert";
       "this intervention does not postulate a patient"s consent"')).
```

The source information is in *def* and is kept for informational purpose. What *Lexdis* uses are the example (*exlex*) and definition (*deflex*) lists that are produced by filtering the original sentences through a stop list to remove frequent words. The synset of the previous example is:

```
% s(synset_id,word_number,'word',synset type, sense number, tag count).
s(202627934,1,'necessitate',v,1,12).
s(202627934,2,'ask',v,6,0).
s(202627934,3,'postulate',v,3,0).
s(202627934,4,'need',v,1,110).
```

```
s(202627934,5,'require',v,1,144).
s(202627934,6,'take',v,14,21).
s(202627934,7,'involve',v,4,12).
s(202627934,8,'call for',v,2,21).
s(202627934,9,'demand',v,2,22).
```

All those verbs are linked with the synset in question, whose definition is "require as useful, just, or proper". Here are two entries that are to be found in *semdic.pl*:

```
mono(lem(necessitate), ori(wn), idnum('necessitate%2:42:00::'), pos(v), lab([]), gw([]),
        deflex([necessitate, ask, postulate, need, require, take, involve, 'call for', demand, require,
                useful, proper]),
        exlex([nerve, success, requires, hard, work, job, asks, patience, skill, position, demands,
                personal, sacrifice, dinner, calls, spectacular, dessert, intervention, postulate, patients,
                consent]),
        def('require as useful, just, or proper; "it takes nerve to do what she did";
             "success usually requires hard work"; "this job asks a lot of patience and skill";
             "this position demands a lot of personal sacrifice"; "this dinner calls for a spectacular dessert";
             "this intervention does not postulate a patient\'s consent'")).
mono(lem(ask), ori(wn), idnum('ask%2:42:00::'), pos(v), lab([]), gw([]),
        deflex([necessitate, ask, postulate, need, require, take, involve, 'call for', demand, require,
                useful, proper]),
        exlex([nerve, success, requires, hard, work, job, asks, patience, skill, position, demands,
                personal, sacrifice, dinner, calls, spectacular, dessert, intervention, postulate, patients,
                consent]),
        def('require as useful, just, or proper; "it takes nerve to do what she did";
             "success usually requires hard work"; "this job asks a lot of patience and skill";
             "this position demands a lot of personal sacrifice"; "this dinner calls for a spectacular dessert";
             "this intervention does not postulate a patient\'s consent'")).
```

First of all, the information associated with "necessitate" is verbatim the same as that of "ask", except for their unique *idnum*. Since they belong to the same synset, it cannot be any other way: there is only one definition per synset. The *deflex* list does not only contain the relevant words taken from the definition – relevant words from the definition "require as useful, just, or proper" are "require", "useful" and "proper" – but also all the synonyms from the same synset.

Apart from the WordNet files, the source files are dictionaries from which the lemma and its definition and examples are kept. The structure of the clauses is exactly the same as the WordNet clause (cf. previous table):

```
mono(lem('ask'),ori('ci'),idnum('ci3500'),pos('v'),lab([]),gw([question]),
        deflex(['question','request','answer']),
        exlex(['asked','question','formal','favour','welsh','history','thought','studied','idea','train','departs',['...'],
                'financial','advice','accountant','give','see','id','awkward','problem','opinion','polite','children',
                'permission','leave','dinner','table','solicitor','client','allowed','telephone','call','faulty','goods',
                'returned','original','packaging','babysitters','moved','asking','replacement']),
      def(' to put a question to someone or to request esp an answer from someone')).
```

The pivot is the lemma, to which are associated the origin of the information, a unique ID, the part of speech, a label and guide word list, a definition and an example list and finally the

complete definition. The guide word list is only a CIDE feature. It helps the reader to choose the right word sense. It is a sort of general indicator. The labels in the label lists are a coding system:

```
mono(lem('aa'),[...],lab(['mp']),[...],
        def(' in britain a film that children under are not admitted to see in a cinema')).
mono(lem('unicorn'),[....],lab(['am','my']),[...],
        def(' an imaginary horselike animal told of in stories with one horn growing from its forehead')).
```

“mp” stands for “motion pictures and film production”, “am” for “animal names” and “my” for “mythology and legend”.

As usual, the semantic proximity of any two words is computed via the intersection of their lists of labels, guide words, definitions and examples: the result that comes from the intersection is the list of shared words. The more words they share, the stronger the proximity. Note that the weighting differs from one list to the other. For instance, the sharing of a guide word is more relevant than that of a label.

8. path.pl

WordNet does not only explore synonymy. It encompasses a lot of sundry semantic relationships. As is always the case in WordNet, all the relationships link synsets. The relationships that are used in *Lexdis* are: hypernymy-hyponymy, entailment (holds only for verbs), instance (the first synset is an instance of the second synset – holds only for nouns), meronymy (holds only for nouns) and causality (holds only for verbs). The pivot is the number of the first synset:

```
path(100003993,[100003553, 100002684, 100001930, 100001740]).
path(100004258,[100003553, 100002684, 100001930, 100001740]).
```

The relationship that binds the synsets is not kept because it is not discriminatory in *Lexdis*. This list that follows the pivot contains one or several synsets whose order in the list indicates their hierarchical level vis-à-vis the pivot.

9. Conclusion: Possibilities and Limitations

Words can only be related thanks relevant databases. Accordingly, the way they are structured and the information they contain influence the result of any query.

Lexdis uses the two bilingual dictionaries the Robert-Collins and the Oxford-Hachette. They provide the information of *coll.pl*, *mt.pl* and *indic.pl*. The version of the databases that were used to produce these files dates from 1995. This means for instance that words linked to more recent

technologies – think about "wireless" and "router" or "connexion", "mobile" and "phone", etc. – may not produce a number as high as one may expect. They also remain general dictionaries that do not encompass specific fields. On the other hand, WordNet contains scientific information, namely in the field of biology and medicine. Roget's Thesaurus is the less up-to-date source as one of the most recent version dates from 1911.

Chapter V: Description of *Lexdis*

0. Introduction

This chapter comments on the code of *Lexdis*. I kept all the original annotations. Note that this chapter is not a rewriting of the code in English; it just provides comments on the code if necessary.

1. Compilation of the modules

```
:- dynamic(kill/1).
:- [ semdic,mt,roget,indic,coll,envir,pesi,s,paths].
```

This loads the different databases into the Prolog database. It also declares "kill/1" as dynamic, meaning that its definition may change during the execution of the programme. Its purpose is to list the entries that have been merged (cf. merge mode, chapter 3.5) so that they can be deleted from the database.

2. User interface

```
go :-
    nl,nl,
    protocola(lexdis),
    write('Computing lexical distance...'),
    nl,
    write('A. Michiels, University of Liège'),nl,
    write('-----------------------------------------'),
    nl,
    nl,
    write('Input file?  [stdin. or filename.] --> '),
    read(Input),
    dwith(Input,HandleIn),
    write('Results file? [filename.] --> '),
    read(Output),
    concat(Output,'.lst',Outlist),
    !,
    start(Outlist,HandleIn,Input).
```

"go" starts the programme. First, it opens a protocol file which logs all the interactions with the user in a file named "lexdis". Then, it displays editorial information and asks the user which type of input he/she wants: standard input (i.e. keyboard) or file input. Finally, the user has to provide an output file name. This file will contain the results provided by the programme. The last clause starts the programme.

```
dwith(Input,stdin):- !.

dwith(FileIn,HandleIn):-
                    open(FileIn,read,HandleIn).
```

There are two possibilities: either the user wants the keyboard as standard input or a file. In the first case, the user entered "stdin", which makes the programme "cut" ("!"): it

will never look at the following clause should it fail. In the second case, the user entered a file name: the programme opens a stream (HandleIn) associated with the provided file name.

3. Starting the programme

3.1. Input from the keyboard (standard input)

```
start(Outlist,_,stdin) :-
    open(Outlist,write,Lists),
    statistics(cputime,BT),
    recorda(time,BT),
    repeat,
    nl,nl,
    write('Please enter the pair of items as follows:
        [word1,pos1,word2,pos2,m:Mode]. or
[word1,pos,word2,m:Mode].') ,nl,
    write('or
[word1,pos1,word2,pos2,w:MW,m:Mode]. '),
    nl,
    write('(where Mode is either g(lobal:words) or
l(ocal:wordsenses),
           pos1, pos2 and pos are either adj, adv, n
or v '), nl,
    write('and MW is an integer expressing the
required minimum connectivity threshold'),nl,
    write('or t or t3 to get the top(or top 3)pairings
only)'),
    nl,
 write('or friends(word,pos,w:MinimumWeight)'),nl,
 write('or t(word,v,arg1,n,arg2,n)'),nl,
    write('or show(word,pos,spec:data).
          where spec is either gw,lb,df or ex
          and data is a label, guide word or word'),nl,
    write('or show(Idnum).
           where Idnum is an identifier returned by the
system'),
    nl,
    write('or finally nadamas. to exit --> '),
    (   recorded(weights,max(0),_)-> true;
recorda(weights,max(0),_)),
  % we start with maximum weight (re)set to zero
    read(Query),
    nl(Lists),nl(Lists),write(Lists,'QUERY: '),
    write(Lists,Query),nl(Lists),nl(Lists),
    dealwith(Query,Lists),
    fail.
```

First, the programme opens a stream linked with the file name provided by the user. It also stores the cpu time the programme has been running so far to be able to compute how long the programme took to compute lexical proximity. Then it enters a typical repeat-fail loop: it always fails so that when it is finished, the user has the possibility to enter a new query. The beginning of the loop presents the user with all the possible choices. The fail-loop also makes Prolog look for all the solutions to the user's query.

3.2. Input from a file

```
start(Outlist,HandleIn,Input) :-
    Input \= stdin,
    open(Outlist,write,Lists),
    nl,
    statistics(cputime,BT),
    recorda(time,BT),
    repeat,
    (   recorded(weights,max(0),_)-> true;
recorda(weights,max(0),_)),
    read(HandleIn,Query),nl,nl,
    write(Query),nl,nl,
    nl(Lists),nl(Lists),write(Lists,'QUERY: '),
    write(Lists,Query),nl(Lists),nl(Lists),
    dealwith(Query,Lists),
    fail.
```

Instead of reading the query from the keyboard, it reads it from the file associated with the stream.

4. Analyzing and Expanding the Query

```
dealwith(nadamas,L) :-
    nl,
    recorded(time,BT),
    statistics(cputime,ET),
    TimeUsed is ET-BT,
    nl(L), write(L,'cputime : '), write(L,TimeUsed),
nl(L),
    nl, write('cputime : '), write(TimeUsed), nl,nl,
    write('End of input... Always glad to be able to
help... Bye!'),nl,nl,
    noprotocol,
    close(L),
    open(tokill,write,Out),
    tell(Out),
    (listing(kill/1) -> true; true),
% we do not necessarily produce kill clauses!
    tell(user),
    close(Out),
    abort.
```

This predicate makes it possible to get out of the programme when the user enters the query “nadamas”. First, it displays the cpu time, which is the total execution time minus the time it took to load the databases. Then it closes the protocol and produces a kill file if need be.

```
dealwith([Word1,Pos,Word2],L) :-
    dealwith([Word1,Pos,Word2,
    Pos,w:none,m:g,noadjust],L).
% no minimum weight specified: we set it to 'none',
% no mode specified: set to global
% + shared POS
```

```
dealwith([Word1,Pos1,Word2,Pos2],L) :-
    Pos2 \= w:W,
    Pos2 \= m:M,
    dealwith([Word1,Pos1,Word2,Pos2,w:none,m:g,noadjust],L).

% we supply global mode and weight is set to 'none'
```

```
dealwith([Word1,Pos,Word2,m:Mode],L) :-
    dealwith([Word1,Pos,Word2,Pos,w:none,m:Mode,noadjust],L).

% no minimum weight specified : we set it to 'none'
% + shared POS
```

```
dealwith([Word1,Pos1,Word2,Pos2,m:Mode],L) :-
    Pos2 \= w:W,
    dealwith([Word1,Pos1,Word2,Pos2,w:none,m:Mode,noadjust],L).

% no minimum weight specified : we set it to 'none'
```

```
dealwith([Word1,Pos1,Word2,Pos2,w:MW,m:Mode],L) :-
dealwith([Word1,Pos1,Word2,Pos2,w:MW,m:Mode,noadjust],L).

% specified MW
```

```
dealwith(t(Target,Arrow1,Arrow2),L) :-
    dealwith(t(Target,v,Arrow1,n,Arrow2,n),L).

% triplets : ArgBearer,Arg1,Arg2 e.g. t(wear,tie,watch)
```

5. Global Mode

5.1. Minimum Weight

```
% global mode : we relate lexical items not word
% senses

% here we are not interested in word senses
% specified by a given idnum
% idnum is allowed an 'existential' reading in the
% basic set of clauses
% the distance is measured as between words
% regrouping all their distinct wordsenses

% we collect the relevant data in all relevant
% semdic entries

dealwith([Word1,Pos1,Word2,Pos2,w:MinimumWei
ght,m:g,noadjust],Output) :-

setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),

Def^Dic^Idnum^Lablist1^Gwlist1^Deflex1^Exlex1^(
mono(lem(Word1),
          ori(Dic),
          idnum(Idnum),
          pos(Pos1),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def)   )),
        DL1),

   setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),

Def^Dic2^Idnum2^Lablist2^Gwlist2^Deflex2^Exlex2
^(mono(lem(Word2),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos2),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def)   )),
        DL2),

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

% we distribute the data just collected : labels,
% guide words, definition core items, example core
% items

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),



collmeet(Word1,Pos1,Word2,Pos2,
          Collweight),

envirmeet(Word1,Word2,Envirweight),

(Pos1=Pos2 -> wordnetmeet(Word1,
        Word2,Pos1,WordNetweight);
                WordNetweight=0),

% weight assignation to collocate field
% sharing, Roget's category sharing,
% indicator sharing, collocate sharing,
% sharing of environment
```

```
compute(lb,L1,L2,ResLabs,WeightLabs),
 compute(gw,G1,G2,ResGws,WeightGws),

% computing the weight to be assigned to
% both labels and guide words

cw(def,Word1,Word2,D1,D2,ResDeflex,
        WeightDeflex),
cw(ex,Word1,Word2,E1,E2,ResExlex,
         WeightExlex),

% idem for definition core items and
% example core items

metameet(Word1,Word2,Metaweight),

rogetmeet(Word1,Word2,Rogetweight),

indicmeet(Word1,Pos1,Word2,Pos2,
              Indicweight),
```

```
sumlist([WeightLabs,WeightGws,
             WeightDeflex,WeightExlex,
             Metaweight,Rogetweight,
             Indicweight,Collweight,
             Envirweight,WordNetweight],
            GW),
(MinimumWeight=none -> true; GW >=
                  MinimumWeight),
  % reaching the threshold

  report(Output,global,GW,Word1,Pos1,
        Word2,Pos2,
        ResLabs,WeightLabs,
        ResGws,WeightGws,
        ResDeflex,WeightDeflex,
        ResExlex,WeightExlex,
        Metaweight,Rogetweight,
        Indicweight, Collweight,
        Envirweight,WordNetweight,
        0).
```

In this mode, the user only wants the queries giving a certain weight to be displayed. After gathering all the information available on the two words in the databases, *Lexdis* computes the lexical distance from each source of information, i.e. labels, guide words, definitions, examples, co-occurrence of collocations, Roget's Thesaurus, indicators, collocations, environment, and WordNet's semantic relationships – the latter only if they share the same part of speech. It adds them up and reports the result only if the addition is greater or equal to the minimum threshold.

5.2. Adjusting Proximity According to Lexical Weight

```
dealwith([Word1,Pos1,Word2,Pos2,
            w:MinimumWeight,m:g,adjust],Output) :-

setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),

Def^Dic^Idnum^Lablist1^Gwlist1^Deflex1^Exlex1^(
mono(lem(Word1),
           ori(Dic),
           idnum(Idnum),
           pos(Pos1),
           lab(Lablist1),
           gw(Gwlist1),
           deflex(Deflex1),
           exlex(Exlex1),
           def(Def)   )),
         DL1),

setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
        Def^Dic2^Idnum2^Lablist2^Gwlist2^
        Deflex2^Exlex2^(mono(lem(Word2),
```

```
Def^Dic2^Idnum2^Lablist2^Gwlist2^Deflex2^
    Exlex2^(mono(lem(Word2),
           ori(Dic2),
           idnum(Idnum2),
           pos(Pos2),
           lab(Lablist2),
           gw(Gwlist2),
           deflex(Deflex2),
           exlex(Exlex2),
           def(Def)   )),
       DL2),

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

% we distribute the data just collected : labels,
% guide words, definition core items, example core
% items
```

```
   flatten(Lab1,L1),
   flatten(Lab2,L2),
   flatten(Gw1,G1),
   flatten(Gw2,G2),
   flatten(Def1,D1),
   flatten(Def2,D2),
   flatten(Ex1,E1),
   flatten(Ex2,E2),

   compute(lb,L1,L2,ResLabs,WeightLabs),
   compute(gw,G1,G2,ResGws,WeightGws),

% computing the weight to be assigned to both
% labels and guide words

cw(def,Word1,Word2,D1,D2,ResDeflex,
            WeightDeflex),
cw(ex,Word1,Word2,E1,E2,ResExlex,
            WeightExlex),
% idem for definition core items and example
% core items

metameet(Word1,Word2,Metaweight),
rogetmeet(Word1,Word2,Rogetweight),

indicmeet(Word1,Pos1,Word2,Pos2,
              Indicweight),

collmeet(Word1,Pos1,Word2,Pos2,
               Collweight),

envirmeet(Word1,Word2,Envirweight),

(Pos1=Pos2 -> wordnetmeet(Word1,Word2,
                    Pos1,WordNetweight);
                     WordNetweight=0),
```

```
% weight assignation to collocate field sharing,
% Roget's category sharing,
% indicator sharing, collocate sharing, sharing
% of environment, WordNet
% path sharing (the latter only if the 2 items
% share POS)

sumlist([WeightLabs,WeightGws,
           WeightDeflex,WeightExlex,
           Metaweight,Rogetweight,
           Indicweight,Collweight,
           Envirweight,WordNetweight],
           GW),

% adjusting weights according to lexical
% weight : remmed or remmable
% if no threshold is specified, neg weights
% also recorded

adjustweight(Word1,Pos1,Word2,Pos2,
                    Adjust),
NW is GW - Adjust,
(MinimumWeight=none -> true;
         NW >= MinimumWeight),

report(Output,global,NW,Word1,Pos1,
        Word2,Pos2,
        ResLabs,WeightLabs,
        ResGws,WeightGws,
        ResDeflex,WeightDeflex,
        ResExlex,WeightExlex,
        Metaweight,Rogetweight,
        Indicweight,Collweight,
        Envirweight,WordNetweight,
        Adjust).
```

The only difference with the previous mode is that it adjusts the result according to the lexical weight of the query (cf. *pesi.pl*).

6. Local Mode

6.1. Top Pairs

```
% local mode : we relate lexical items, individuating
% word senses in our lexical data bases
% the Idnum field no longer receives an existential
% reading in the setof clauses
% items are paired Idnum to Idnum,
% which are supposed to be assigned to word
% senses
```

```
% we show only the top pairings ('w:t' option)

dealwith([Word1,Pos1,Word2,Pos2,w:t,m:l,
             noadjust],Output) :-

setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),
```

```
Def^Dic^Lablist1^Gwlist1^Deflex1^
     Exlex1^(mono(
       lem(Word1),
          ori(Dic),
          idnum(Idnum1),
          pos(Pos1),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def) )),
        DL1) ,

setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
       Def^Dic2^Lablist2^Gwlist2^Deflex2^
        Exlex2^(mono(lem(Word2),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos2),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def) )),
       DL2) ,

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),

  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),

cw(def,Word1,Word2,D1,D2,ResDeflex,
     WeightDeflex),
  cw(ex,Word1,Word2,E1,E2,ResExlex,
               WeightExlex),
sumlist([WeightLabs,WeightGws,WeightDeflex,
           WeightExlex],W),

% each time we get a new WS (Word Sense) pair
% we see if its weight is inferior to the recorded
% maximum
% if it is, we simply drop it and wait for the intrinsic
% failure built in the dealwith loop to provide other
% pairs

% and record the new value as max
% if it comes up to the maximum but does not
% exceed it, we simply record it as a pair to
% remember

% if it exceeds the recorded max, we 'forget' the
% recorded max,


  ( ( recorded(weights,max(Max),_),
     Max > W
     )
     ->
     true
     ;
     (recorded(weights,max(Max),Ref),
      (   W=Max -> recorda(res,
str([Output,local,W,Word1,Idnum1,Pos1,

Word2,Idnum2,Pos2,ResLabs,WeightLabs,

ResGws,WeightGws,ResDeflex,WeightDeflex,
                              ResExlex,WeightExlex]),_)
                ;
                (   erase(Ref),
                recorda(weights,max(W),_),
                deleteref1(Max),
                recorda(res,
str([Output,local,W,Word1,Idnum1,Pos1,
                              Word2,Idnum2,Pos2,
                              ResLabs,WeightLabs,
                              ResGws,WeightGws,
                              ResDeflex,WeightDeflex,
                              ResExlex,WeightExlex]),_)
                )
     )
    )
   ).


% when all pairs have been examined and, if
% need be, recorded
% we can gather them all together with a setof
% clause on the recorded pairs
% and report on the results
% we end up by cleaning, i.e. erasing all
% recorded information

dealwith([W1,P1,W2,P2,w:t,m:l,noadjust],
             Output) :-
      setof( Args,
            recorded(res,str(Args),_),
            ListArgs),
      reportlist(ListArgs),
      eraseall(weights),
      eraseall(res).
```

```
% we also forget all the pairs we have recorded
% so far,since they are now dethroned by the pair
% exhibiting the new max
% we can then record the new winning pair and
% expect the intrinsic failure built in the dealwith
% loop to provide other candidates
```

```
reportlist([]).
reportlist([Args|MoreArgs]) :-
        report(Args),
         reportlist(MoreArgs).
```

This mode only displays the word sense(s) with the greatest lexical weight. The major difference with the global mode is that each word sense is taken into account. This translates to the "idnum" not being preceded by the existential quantifier:

- **global mode**: Def^Dic^**Idnum**^Lablist1^Gwlist1^Deflex1^Exlex1^(mono(lem(Word1);
- **local mode**: Def^Dic^Lablist1^Gwlist1^Deflex1^Exlex1^(mono(lem(Word1).

This entails that *Lexdis* computes lexical proximity between each word sense of each word.

6.2. Minimum Weight

```
dealwith([Word1,Pos1,Word2,Pos2,
            w:MW,m:l,noadjust],Output) :-
   MW \= t, MW \= m, MW \= t3,
   member(Pos1,[n,v,adj,adv]),
   setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),

Def^Dic^Lablist1^Gwlist1^Deflex1^Exlex1^(mono(
lem(Word1),
          ori(Dic),
          idnum(Idnum1),
          pos(Pos1),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def) )),
        DL1) ,

   49etoff(data(Lablist2,Gwlist2,Deflex2,Exlex2),

Def^Dic2^Lablist2^Gwlist2^Deflex2^Exlex2^(mono
(lem(Word2),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos2),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def) )),
       DL2) ,

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),

  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),

cw(def,Word1,Word2,D1,D2,ResDeflex,
    WeightDeflex),

cw(ex,Word1,Word2,E1,E2,ResExlex,WeightExlex),

sumlist([WeightLabs,WeightGws,
          WeightDeflex,WeightExlex],W),

  (MW=none -> true; W >= MW),

  report([Output,local,W,Word1,Idnum1,Pos1,
       Word2,Idnum2,Pos2,
       ResLabs,WeightLabs,
       ResGws,WeightGws,
       ResDeflex,WeightDeflex,
       ResExlex,WeightExlex]).
```

The process is the same as for the global mode: the result is recorded only if it is greater or equal to the minimum weight requested by the user.

6.3. Idnum instead of Part of Speech for the First Item

```
dealwith([Word1,Idnum1,Word2,Pos2,
          w:MW,m:l,noadjust],Output) :-
   MW \= t, MW \= m, MW \=t3,
   Idnum1 \= n, Idnum1 \= v , Idnum1 \= adv,
   Idnum1 \= adj,
   setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),
Def^Dic^Lablist1^Gwlist1^Deflex1^Exlex1^(mono(l
em(Word1),
          ori(Dic),
          idnum(Idnum1),
          pos(Pos1),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def) )),
        DL1) ,

   setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
Def^Dic2^Lablist2^Gwlist2^Deflex2^Exlex2^(mono(
lem(Word2),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos2),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def) )),
       DL2) ,

  Word1 \= Word2,
  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),

  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),


  cw(def,Word1,Word2,D1,D2,ResDeflex,
       WeightDeflex),

  cw(ex,Word1,Word2,E1,E2,ResExlex,
       WeightExlex),

  sumlist([WeightLabs,WeightGws,
           WeightDeflex,WeightExlex],W),

   (MW=none -> true; W >= MW),

  report([Output,local,W,Word1,Idnum1,Pos1,
      Word2,Idnum2,Pos2,
      ResLabs,WeightLabs,
      ResGws,WeightGws,
      ResDeflex,WeightDeflex,
      ResExlex,WeightExlex]).
```

Instead of providing *Lexdis* with a part of speech for the first word, it is possible to enter the identification number of a specific word sense so that *Lexdis* only deals with it.

7. Showing the Relevant Semdic Entry

```
dealwith(show(Word,Pos,Spec:Data),L) :-
       show(Word,Pos,Spec:Data,L).

dealwith(show(Idnum),L) :-
       show(Idnum,L).

dealwith(show(Word,Pos),L) :-
       show(Word,Pos,L).
```

This mode displays all the information available in *semdic* on the word requested by the user.

8. Predicates Used in Local and Global Modes

8.1. Collect

```
collect([],[],[],[],[]).
collect([data(L,G,D,E)|MoreData],
        [L|MoreL], [G|MoreG],
        [D|MoreD],[E|MoreE]) :-

collect(MoreData,MoreL,MoreG,MoreD,MoreE).

% redistributing the data into the 4 relevant lists:
% Labels, Guide Words, Words_in_def,
Words_in_ex
% merge mode
```

```
collect([],[],[],[],[],[]).
collect([data(L,G,D,E,F)|MoreData],
        [L|MoreL], [G|MoreG],
        [D|MoreD],[E|MoreE],[F|MoreF]) :-

collect(MoreData,MoreL,MoreG,MoreD,MoreE,
          MoreF).

% redistributing the data into the 5 relevant lists:
% Labels, Guide Words, Words_in_def,
Words_in_ex, DefStrings
```

The left column displays “collect” used in merge mode, the right column the normal version. When *Lexdis* retrieves lexical information from *semdic*, it is put in a single list in the same order: label, guide word, definition and example (plus definition string in normal mode). This predicate sorts the information into 4 or 5 different lists.

8.2. Compute

```
compute(lb,L1,L2,R,Weight) :-
       (   is_set(L1) -> Set1=L1 ;
list_to_set(L1,Set1)),
       (   is_set(L2) -> Set2=L2 ;
list_to_set(L2,Set2)),
       intersection(Set1,Set2,R),
       length(R,Len),
       Weight is Len*4.

% weighting for labels: each shared label
% is worth 4.
```

```
compute(gw,L1,L2,R,Weight) :-
       (   is_set(L1) -> Set1=L1 ;
list_to_set(L1,Set1)),
       (   is_set(L2) -> Set2=L2 ;
list_to_set(L2,Set2)),
       intersection(Set1,Set2,R),
       length(R,Len),
       Weight is Len*12.

% weighting for guide words: each shared
% guide word is worth 12
```

It computes the lexical proximity of labels and guide words. The code is also identical: the differences lie in the parameter of the predicate (“lb” vs. “gw”) and in the weight (guide words are worth more).

8.3. Cw

```
cw(def,Word1,Word2,L1,L2,Res,Weight) :-
       Word1 \= Word2,
       (   is_set(L1) -> Set1=L1 ;
                         list_to_set(L1,Set1)),
       (   is_set(L2) -> Set2=L2 ;
                         list_to_set(L2,Set2)),
       (member(Word1,Set2) -> Bonus1=20 ;
                        Bonus1=0 ),
```

```
% bonus is granted when the words 'interdefine',
% i.e. A is used in the definition of B, or vice-versa

       (member(Word2,Set1) -> Bonus2=20 ;
                                Bonus2=0 ),

   intersection(Set1,Set2,Res),
       length(Res,Len),
```

```
    (Len <3 -> DefLen is Len*2 ; DefLen is Len*5),

    % we increase the weight if we believe
    % that the sharing exceeds a threshold
    % at which it could still be regarded as
    % incidental (set to 2)
    sumlist([DefLen,Bonus1,Bonus2],Weight).

cw(ex,Word1,Word2,L1,L2,Res,Weight) :-
      Word1 \= Word2,
    (   is_set(L1) -> Set1=L1 ;
                      list_to_set(L1,Set1)),
    (   is_set(L2) -> Set2=L2 ;
                       list_to_set(L2,Set2)),
    (member(Word1,Set2) -> Bonus1=5 ;
                           Bonus1=0 ),
    (member(Word2,Set1) -> Bonus2=5 ;
                           Bonus2=0 ),
    intersection(Set1,Set2,Res),
    length(Res,Len),
    sumlist([Len,Bonus1,Bonus2],Weight).
```

It computes the lexical proximity of the definitions and examples. In both cases, a bonus is granted if the first word is part of the definition or example list of the second word and vice versa. The following clauses are only used in merge mode:

```
cw(def,Word,Word,L1,L2,Union,Weight) :-
    (   is_set(L1) -> Set1=L1 ;
                      list_to_set(L1,Set1)),
    (   is_set(L2) -> Set2=L2 ;
                      list_to_set(L2,Set2)),
    (pick(Word,Set1,SS1) -> NS1=SS1;
                            NS1=Set1),
    (pick(Word,Set2,SS2) -> NS2=SS2;
                            NS2=Set2),
    union(NS1,NS2,Union),
    intersection(NS1,NS2,Res),
    length(Res,Len),
    (Len <3 -> Weight is Len*2 ; Weight is Len*3).

w(ex,Word,Word,L1,L2,Union,Weight) :-
    (   is_set(L1) -> Set1=L1 ;
                      list_to_set(L1,Set1)),
    (   is_set(L2) -> Set2=L2 ;
                       list_to_set(L2,Set2)),
    (pick(Word,Set1,SS1) -> NS1=SS1;
                            NS1=Set1),
    (pick(Word,Set2,SS2) -> NS2=SS2;
                            NS2=Set2),
    union(NS1,NS2,Union),
    intersection(NS1,NS2,Res),
    length(Res,Weight).
```

We make sure that the word is not part of the definition or example list. If it is, it is removed.

8.4. Adjustweight

```
% working out decrease factor for heavy items
% adjustweight(Word1,Pos1,Word2,Pos2,Adjust)
% calls on pesi db, with clauses such as:
% w(fox, v, 189).

adjustweight(Word1,Pos1,Word2,Pos2,Adjust) :-
 (w(Word1,Pos1,Weight1) -> W1 is Weight1 ;
                           W1 is 1),
 (w(Word2,Pos2,Weight2) -> W2 is Weight2 ;
                           W2 is 1),

 Both is W1+W2,
 ( Both < 200 ->  Adjust is 0 ;
                  Adjust is Both//200).
```

If the combined lexical weight of both words exceeds 199, it is divided by 200 and later (cf. global or local mode) subtracted from the result.

8.5. Global Report

```
report(FO,global,W,Word1,Pos1,
       Word2,Pos2,
       ResLabs,WeightLabs,
       ResGws,WeightGws,
```

```
    ResDeflex,WeightDeflex,
    ResExlex,WeightExlex,Metaweight,Rogetweight,
    Indicweight,Collweight,Envirweight,WordNetweight,Adjust) :-

    ( W > 0 ->
(nlb(FO),writeb(FO,Word1),writeb(FO,' with POS='), writeb(FO,Pos1),
 writeb(FO,' is related to '), writeb(FO,Word2), writeb(FO,' with POS='),
 writeb(FO,Pos2), writeb(FO,' with weight='),
 writeb(FO,W),writeb(FO,' as follows: '), nlb(FO),
      (WeightLabs > 0 ->
                  (writeb(FO,'Shared Labels: '),
                   writeb(FO,ResLabs),
                   writeb(FO,' -> weight: '),
                   writeb(FO,WeightLabs),nlb(FO)) ; true),

      (WeightGws > 0 ->
                  (writeb(FO,'Shared Guide Words: '),
                   writeb(FO,ResGws),
                   writeb(FO,' -> weight: '),
                   writeb(FO,WeightGws), nlb(FO)) ; true),

      (WeightDeflex  > 0 ->
                (writeb(FO,'Shared words in definition: '),
                   writeb(FO,ResDeflex),
                   writeb(FO,' -> weight: '),
                   writeb(FO,WeightDeflex), nlb(FO)) ; true),

      (WeightExlex > 0 ->
                   (writeb(FO,'Shared words in examples: '),
                    writeb(FO,ResExlex),
                    writeb(FO,' -> weight: '),
                    writeb(FO,WeightExlex), nlb(FO)) ; true),
      (Metaweight > 0 ->
                   (writeb(FO,'Cooccurrence in collocate lists '),
                    writeb(FO,' -> weight: '),
                    writeb(FO,Metaweight), nlb(FO)) ; true),
      (Rogetweight > 0 ->
                (writeb(FO,'Cooccurrence in Roget\'s thesaurus '),
                    writeb(FO,' -> weight: '),
                    writeb(FO,Rogetweight), nlb(FO)) ; true),
      (Indicweight > 0 ->
              (writeb(FO,'Cooccurrence in R/C-Oxf/Hach indic db '),
                    writeb(FO,' -> weight: '),
                    writeb(FO,Indicweight), nlb(FO)) ; true),
      (Collweight > 0 ->
           (writeb(FO,'Cooccurrence in R/C-Oxf/Hach collocates db '),
            writeb(FO,' -> weight: '),
                    writeb(FO,Collweight), nlb(FO)) ; true),
      (Envirweight > 0 ->
        (writeb(FO,'Cooccurrence in R/C-Oxf/Hach extended lemma db '),
                    writeb(FO,' -> weight: '),
                    writeb(FO,Envirweight), nlb(FO)) ; true),
      (WordNetweight > 0 ->
        (writeb(FO,'Sharing WordNet path '),
                    writeb(FO,' -> weight: '),
                    writeb(FO,WordNetweight), nlb(FO)) ; true),
```

```
        (Adjust > 0 ->
                    (writeb(FO,'Penalty for Heavy Lexical Items '),
                     writeb(FO,' -> neg_weight: - '),
                     writeb(FO,Adjust), nlb(FO)) ; true)
          )
          ;
    (
    nlb(FO), writeb(FO,Word1), writeb(FO,' with POS='), writeb(FO,Pos1),
    writeb(FO,' is ***not*** related to '),
    writeb(FO,Word2), writeb(FO,' with POS='),
    writeb(FO,Pos2), writeb(FO,' (Adjusted Weight = '),
    writeb(FO,W), writeb(FO,').')
      )
      ).
```

The global report writes the results of the query in global mode to a file and on the screen. It details the source of the result.

8.6. Local Report

```
report([FO,local,W,Word1,Idnum1,Pos1,
        Word2,Idnum2,Pos2, ResLabs,WeightLabs,
        ResGws,WeightGws, ResDeflex,WeightDeflex,
        ResExlex,WeightExlex]) :-

        ( W > 0 ->
             (nlb(FO),
              writeb(FO,Word1), writeb(FO,' with POS='), writeb(FO,Pos1), writeb(FO,' and Idnum='),
              writeb(FO,Idnum1), writeb(FO,' is related to '),writeb(FO,Word2),writeb(FO,' with POS='),
              writeb(FO,Pos2),writeb(FO,' and Idnum='),writeb(FO,Idnum2),writeb(FO,' with weight='),
              writeb(FO,W),writeb(FO,' as follows: '), nlb(FO),
              (WeightLabs > 0 ->
                        (writeb(FO,'Shared Labels: '),
                         writeb(FO,ResLabs),
                         writeb(FO,' -> weight: '),
                         writeb(FO,WeightLabs),nlb(FO)) ; true),

              (WeightGws > 0 ->
                        (writeb(FO,'Shared Guide Words: '),
                         writeb(FO,ResGws),
                         writeb(FO,' -> weight: '),
                         writeb(FO,WeightGws), nlb(FO)) ; true),

              (WeightDeflex  > 0 ->
                        (writeb(FO,'Shared words in definition: '),
                         writeb(FO,ResDeflex),
                         writeb(FO,' -> weight: '),
                         writeb(FO,WeightDeflex), nlb(FO)) ; true),

              (WeightExlex > 0 ->
                        (writeb(FO,'Shared words in examples: '),
                         writeb(FO,ResExlex),
                         writeb(FO,' -> weight: '),
```

```
                        writeb(FO,WeightExlex), nlb(FO)) ; true)
                    ;
        true)).
```

It writes the result of a query in local mode to a file and on the screen.

8.7. Metameet

```
metameet(W1,W2,Weight) :-
      W1 @< W2,
      mt(W1,List),
      member([W2,Cooc],List),
      (
(W1=person;W2=person;W1=object;W2=object) ->
      % reduction due to high frequency
      % and poor semantic discriminatory power
            % of 'person' and 'object'
                  Weight is Cooc // 16;
                  Weight is Cooc),
            !.

metameet(W1,W2,Weight) :-
      W1 @> W2,
      mt(W2,List),
      member([W1,Cooc],List),
      (
(W1=person;W2=person;W1=object;W2=object),
                  Weight is Cooc // 16;
                  Weight is Cooc),
            !.

metameet(_,_,0).

% connectedness through cooccurrence
% in Robert/Collins-Oxford/Hachette collocate lists
% hypothesis of connectedness through shared
% belonging to coll lists put forward by Montemagni
% et al.

% cf. Montemagni, S., Federici, S. and Pirrelli,V.
% 1996.
% 'Example-based Word Sense Disambiguation: a
% Paradigm-driven Approach',
% Euralex'96 Proceedings, Göteborg University,
% 151-160.

% the cooccurrence lists are assigned as early as
% possible in the alphabetical ranking of the lexical
% items
% - it is therefore the 'smaller' word that should be
% explored an mt line looks like the following:

/*
mt(digestion,[[growth,1],[machine,1],[mind,1],
                  [movement,1],[reaction,1],
                  [recovery,1],[stomach,4]]).
*/
% this means that the word 'digestion' cooccurs 1
% time with 'growth'
% in a collocate list ... and 4 times with 'stomach'
% the sharing of 'digestion' with a word preceding
% 'digestion' should be looked for under that word
```

8.8. Wordmeet

```
wordnetmeet(W1,W2,Pos,Poids) :-
            findall(Weight,
                  (s(Synset1,_WN1,W1,Pos,_SN1,_),
                   s(Synset2,_WN2,W2,Pos,_SN2,_),
                   wnpsh(Synset1,Synset2,Weight)),
                   Weights),
              sort(Weights,Sorted),


              (Sorted=[GW|_] -> Poids is -GW;
                              Poids is 0).

wnpsh(S,S,-32) :- !.

wnpsh(S1,S2,Weight) :-
   path(S1,P1),
/*
connectedness through the sharing of WordNet
paths;
the WordNet 's' predicate yields the Synset to
which a Word-Pos pair belongs;
example for the noun 'suppressor' :
s(105441468,1,'suppressor',n,2,0).
*/

% we keep the highest weight as we are working
% with lexical items not word senses

% same Synset (WordNet synonymy)
```

```
    path(S2,P2),
    fcn(S1,S2,P1,P2,Weight).                % first common node on the paths

fcn(_,_,[A|_],[A|_],-12) :- !.              % same father

fcn(A,_,_,[A|_],-10) :- !.                  % A is B's father or vice-versa
fcn(_,B,[B|_],_,-10) :- !.

fcn(A,_,_,[_,A|_],-8) :- !.                 % A is B's grandfather or vice-versa
fcn(_,B,[_,B|_],_,-8) :- !.

fcn(_,_,[A|_],[_,A|_],-6) :- !.             % A is B's grandfather or vice-versa
fcn(_,_,[_,A|_],[A|_],-6) :- !.             % the grandfather of the other or vice-versa

fcn(_,_,[_,B|_],[_,B|_],-5) :- !.           % same grandfather

fcn(A,_,_,[_,_,A|_],-3) :- !.               % A is B's greatgrandfather or vice-versa
fcn(_,B,[_,_,B|_],_,-3) :- !.

fcn(_,_,[_,B|_],[_,_,B|_],-2) :- !.         % the grandfather of one is etc...
fcn(_,_,[_,_,B|_],[_,B|_],-2) :- !.

fcn(_,_,[_,_,A|_],[_,_,A|_],-1) :- !.       % same ancestor ... not too high up

fcn(_,_,_,_,0).                             % not related or related too far up
```

8.9. Rogetmeet

```
rogetmeet(W1,W2,Weight) :-
        W1 @< W2,
        r(W1,W1list),
        r(W2,W2list),
        i_roget(W1list,W2list,Commonlist),
        sumlist(Commonlist,Weight),
        !.

rogetmeet(W1,W2,Weight) :-
        W2 @< W1,
        r(W1,W1list),
        r(W2,W2list),
        i_roget(W2list,W1list,Commonlist),
        sumlist(Commonlist,Weight),
        !.

rogetmeet(_,_,0).

i_roget([],_,[]).

i_roget([[Pos,Cat,Sub1,Sub2]|Tail],
            Otherlist,[Weight|Tail2]) :-


        member([P,Cat,Othersub1,Othersub2],
```

```
/*
connectedness through the sharing of Roget's
categories;
three levels of delicacy in thesaurus organisation;
we retrieve the list of categories associated with the
two items and then we compute their intersection,
accumulating the weights according to type of
category matched;
 the check on order of the two items (W1 and W2)
ensures that the match is always exactly the same
independent of the ordering of the two items in the
query cat should rogetmeet dog exactly like dog
rogetmeets cat;
we ensure this by always matching cat-dog, never
dog-cat */


                                            % computing the intersection

                                            % we leave POS as a variable, desinstantiating it to
                                            % P as the POS field should not be relevant to
                                            % thesauric organisation, although it is (for
                                            % instance, in Roget's itself!)
```

```
                Otherlist),
        !,
        irogetweight(Sub1,Sub2,Othersub1,
                     Othersub2,Weight),
        i_roget(Tail,Otherlist,Tail2).

i_roget([A|Tail],Otherlist,Tail2) :-
        i_roget(Tail,Otherlist,Tail2).

irogetweight(A,B,A,B,3) :-   !.
irogetweight(A,_,A,_,2) :-   !.
rogetweight(_,_,_,_,1) :-   !.
```

% all three categories
% cat and sub-cat
% only broadest cat

8.10. Indicmeet

```
indicmeet(Word1,Pos1,Word2,Pos2,IndicWeight) :-
    findall(Indic,
            ind(lemma(Word1),
                pos(Pos1),indic(Indic)),
            IndicList1),
    findall(Indic2,
           nd(lemma(Word2),
               pos(Pos2),indic(Indic2)),
           IndicList2),
    flatten(IndicList1, IL1),
    flatten(IndicList2, IL2),
    (is_set(IL1) -> Set1=IL1 ; list_to_set(IL1,Set1)),
    (is_set(IL2) -> Set2=IL2 ; list_to_set(IL2,Set2)),
    (member(Word1,Set2) -> Bonus1=5 ;
                           Bonus1=0 ),
    (member(Word2,Set1) -> Bonus2=5 ;
                           Bonus2=0 ),
    intersection(Set1,Set2,Res),
    length(Res,Len),
    (Len <3 -> IW is Len*2; IW is Len*4),
% we prioritize 'non-incidental' sharing (threshold
% set to 3)
    sumlist([IW,Bonus1,Bonus2],IndicWeight),
    !.

indicmeet(_W,_P,_W2,_P2,0).
```

It computes connectedness through indicator sharing.

8.11. Collmeet

```
collmeet(Word1,Pos1,Word2,Pos2,CollWeight) :-
    findall(Coll,
           coll(lemma(Word1),pos(Pos1),coll(Coll)),
           CollList1),
    findall(Coll2,
           coll(lemma(Word2),pos(Pos2),coll(Coll2)),
           CollList2),
    flatten(CollList1, CL1),
    flatten(CollList2, CL2),
    (is_set(CL1) -> Set1=CL1 ;
              list_to_set(CL1,Set1)),
    (is_set(CL2) -> Set2=CL2 ;
              list_to_set(CL2,Set2)),
    (member(Word1,Set2) -> Bonus1=20 ;
                           Bonus1=0 ),
    (member(Word2,Set1) -> Bonus2=20 ;
                           Bonus2=0 ),
    intersection(Set1,Set2,Res),
    length(Res,Len),
    (Len <3 -> CW is Len*2 ; CW is Len*5),
    sumlist([CW,Bonus1,Bonus2],CollWeight),
    !.

collmeet(_W,_P,_W2,_P2,0).

/*
Connectedness through collocate sharing in
RC/OH collocate data base

db struc (coll.pl) : e.g.

coll(lemma('abandonment'),
     pos(n),
     coll(['property','right'])).

here, contrary to what we get through metameet,
the two items are related if they POSSESS
common elements in their collocate lists;
in metameet it is the copresence within a collocate
list (associated with whatever item) that is
significant
*/
```

8.12. Envirmeet

```
/* Connectedness through envir sharing in RC/OH
envir data base;
db struc (envir.pl) : e.g.
e(hdwd('dative'),envir(['case','ending'])).
the POS are not significant here
*/
envirmeet(Word1,Word2,EnvirWeight) :-
    findall(Envir,
             e(hdwd(Word1),envir(Envir)),
             EnvirList1),
    findall(Envir2,
            e(hdwd(Word2),envir(Envir2)),
            EnvirList2),
    flatten(EnvirList1, EL1),
    flatten(EnvirList2, EL2),
    (is_set(EL1) -> Set1=EL1 ;
             list_to_set(EL1,Set1)),
        (is_set(EL2) -> Set2=EL2 ;
             list_to_set(EL2,Set2)),
    (member(Word1,Set2) -> Bonus1=3 ;
                           Bonus1=0 ),
    (member(Word2,Set1) -> Bonus2=3 ;
                           Bonus2=0 ),
    intersection(Set1,Set2,Res),
    length(Res,Len),
    (Len >1 -> EW is Len; EW is 0),
%  we neglect an envir sharing limited to a single
% element(probably non-significant most of the
% time)
    sumlist([EW,Bonus1,Bonus2],EnvirWeight),
    !.

envirmeet(_W,_W2,0).
```

9. Show

It displays information on what the user requests on the screen and to a file. Each clause deals with a type of information (guide word, label, etc.).

```
show(Word,Pos,gw:GW,L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
        member(GW,Gwlist),
        writeb(L,Word),tabb(L,2), writeb(L,Pos),
tabb(L,2),
        writeb(L,Gwlist),tabb(L,2),writeb(L,Lablist),
tabb(L,2),
        writeb(L,Idnum), tabb(L,2),
        writeb(L,'Def: '), writeb(L,Def),
        nlb(L),nlb(L).
```

```
show(Word,Pos,lb:Lab,L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
        member(Lab,Lablist),
        writeb(L,Word),tabb(L,2), writeb(L,Pos),
tabb(L,2),
        writeb(L,Gwlist),tabb(L,2),writeb(L,Lablist),
tabb(L,2),
         writeb(L,Idnum), tabb(L,2),
        writeb(L,'Def: '), writeb(L,Def),
        nlb(L),nlb(L).
```

```
show(Word,Pos,df:Defel,L) :-
        Defel \= and(Candi),
        Defel \= or(Candi),
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
    exlex(Exlex),def(Def)),
    member(Defel,Deflex),
    writeb(L,Word),tabb(L,2),
    writeb(L,Pos),tabb(L,2),
    writeb(L,Gwlist),tabb(L,2),
    writeb(L,Lablist),tabb(L,2),
    writeb(L,Idnum),tabb(L,2),
    writeb(L,Deflex),nlb(L),
    writeb(L,'Def: '), writeb(L,Def),
    nlb(L),nlb(L).
```

```
show(Word,Pos,df:and(Defellist),L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
        allmembers(Defellist,Deflex),
        writeb(L,Word),tabb(L,2), writeb(L,Pos),
        tabb(L,2),
        writeb(L,Gwlist),tabb(L,2),writeb(L,Lablist),
tabb(L,2),
        writeb(L,Idnum),
tabb(L,2),writeb(L,Deflex),nlb(L),
        writeb(L,'Def: '), writeb(L,Def),
        nlb(L),nlb(L).
```

```
show(Word,Pos,df:or(Defellist),L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
        anymember(Defellist,Deflex),
        writeb(L,Word),tabb(L,2), writeb(L,Pos),
tabb(L,2),
        writeb(L,Gwlist),tabb(L,2),writeb(L,Lablist),
tabb(L,2),
        writeb(L,Idnum),
tabb(L,2),writeb(L,Deflex),nlb(L),
        writeb(L,'Def: '), writeb(L,Def),
        nlb(L),nlb(L).
```

```
show(Word,Pos,ex:Exel,L) :-
        Exel \= and(Candi),
        Exel \= or(Candi),
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
            member(Exel,Exlex),
            writeb(L,Word),tabb(L,2), writeb(L,Pos),
tabb(L,2),
            writeb(L,Gwlist),tabb(L,2),writeb(L,Lablist),
tabb(L,2),
            writeb(L,Idnum),
tabb(L,2),writeb(L,Exlex),nlb(L),
            writeb(L,'Def: '), writeb(L,Def),
            nlb(L),nlb(L).
```

```
show(Word,Pos,ex:and(Exellist),L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
            allmembers(Exellist,Exlex),
            writeb(L,Word),tabb(L,2), writeb(L,Pos),
            tabb(L,2),
            writeb(L,Gwlist),tabb(L,2),
            writeb(L,Lablist),tabb(L,2),
            writeb(L,Idnum),
            tabb(L,2),writeb(L,Exlex),nlb(L),
            writeb(L,'Def: '), writeb(L,Def),
            nlb(L),nlb(L).
```

```
show(Word,Pos,ex:or(Exellist),L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def)),
            anymember(Exellist,Exlex),
            writeb(L,Word),tabb(L,2), writeb(L,Pos),
            tabb(L,2),
            writeb(L,Gwlist),tabb(L,2),
            writeb(L,Lablist),tabb(L,2),
            writeb(L,Idnum),
            tabb(L,2),writeb(L,Exlex),nlb(L),
            writeb(L,'Def: '), writeb(L,Def),
            nlb(L),nlb(L).
```

```
show(Idnum,L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def) ),
            writeb(L,Word),tabb(L,2),
            writeb(L,Pos), tabb(L,2),
            writeb(L,Gwlist),tabb(L,2),
            writeb(L,Lablist),tabb(L,2),
            writeb(L,Idnum),
            tabb(L,2),writeb(L,Exlex),nlb(L),
            writeb(L,'Def: '), writeb(L,Def),
            nlb(L),nlb(L).
```

```
show(Word,Pos,L) :-
        mono(lem(Word),
            ori(Dic),
            idnum(Idnum),
            pos(Pos),
            lab(Lablist),
            gw(Gwlist),
            deflex(Deflex),
            exlex(Exlex),
            def(Def) ),
            writeb(L,Word),tabb(L,2),
            writeb(L,Pos), tabb(L,2),
            writeb(L,Gwlist),tabb(L,2),
            writeb(L,Lablist),tabb(L,2),
            writeb(L,Idnum),
            tabb(L,2),writeb(L,Exlex),nlb(L),
            writeb(L,'Def: '), writeb(L,Def),
            nlb(L),nlb(L).
```

10. Utilities

The following predicates are used in other predicates as utilities.

```
% allmembers(Subset,List)
allmembers([],_).
allmembers([H|Tail],List) :- member(H,List),
allmembers(Tail,List).
% anymember(Candidates,List)
anymember(L,L1) :- intersection(L,L1,[_|_]).
```

```
% writing to both file and standard output
writeqb(FO,X) :- writeq(FO,X), writeq(X).
writeb(FO,X) :- write(FO,X), write(X).
nlb(FO) :- nl(FO), nl.
tabb(FO,V) :- tab(V), tab(FO,V).
```

```
% erasing all items sharing a property
%-------------------------------------

% here : having the same weight
% (recorded as Max)
deleteref1(Max) :-
  recorded(res,
          str([_Output,local,Max,_Word1,
               _Idnum1,_Pos1,
               _Word2,_Idnum2,_Pos2,
               _ResLabs,_WeightLabs,
               _ResGws,_WeightGws,
               _ResDeflex,_WeightDeflex,
               _ResExlex,_WeightExlex]),
          Ref),
  erase(Ref),
  fail.

deleteref1(_Max).
```

```
deleteref2(Max) :-
  recorded(entries,
          mono(lem(_Word),
              ori(Max),
              idnum(_Idnum),
              pos(_Pos),
              lab(_Lablist),
              gw(_Gwlist),
              deflex(_Deflex),
              exlex(_Exlex),
              def(_DefW) ),
          Ref),
  erase(Ref),
  fail.

deleteref2(_Max).
```

```
% erasing recorded info
%----------------------

% erasing all items in a 'box'

eraseall(X) :-
    recorded(X,_,Ref),
    erase(Ref),
    fail.

eraseall(_).

% takes an element out of a list
pick(H,[H|T],T) :- !.
pick(H,[Other|T],[Other|T1]) :- pick(H,T,T1).
```

```
% stores an element if it hasn't been recorded yet
store(X) :- not(recorded(entries,X,_)),
            recorda(entries,X,_).

% ordering of Idnums
myorder(A,B) :-  name(A,ListA), name(B,ListB),
                 orderlist(ListA,ListB).

orderlist([Aprem|Areste],[Aprem|Breste]) :-
              orderlist(Areste,Breste).

orderlist([Aprem|Areste],[Bprem|Breste]) :-
               Aprem < Bprem.
orderlist([],_).
```

11. Merge Mode

11.1. Dealwith

```
% here we are dealing with a single lexeme-pos pair
% possibly giving rise to a whole bunch of entries in semdic
% as semdic takes over four monolinguals describing a similar lexical range
% (cide, cobuild, ldoce, wordnet)
% we investigate which pair of lex-pos/lex-pos
% yields the best match
% the successful pair is deemed to be a good candidate
% for an inter-dic and/or across-dic merge

% the process is meant to be executed recursively
% until the best l-p pair is not good enough to justify a merge
% this should be established through trial and error


dealwith([Word,Pos,Word,Pos,w:m,m:l,noadjust],Output) :-
    w(lem(Word),weight(LW)),
   %  LW = 1,
       % give merge proposals only for suitably heavy or light items
     eraseall(weights),
     eraseall(entries),
     recorda(weights,max(16),_),
       % we start with maximum weight (re)set
       % and make sure we have 'forgotten' data belonging to a previous pair
     dwmerge([Word,Pos,Word,Pos,w:m,m:l,noadjust],Output).
```

11.2. Dwmerge

```
% all info existentially quantified except:
% Word    Pos  Idnum
dwmerge([Word,Pos,Word,Pos,
             w:m,m:l,noadjust],Output) :-

setof(data(Lablist1,Gwlist1,Deflex1,Exlex1,DefW1),
```

```
DefW1^Dic1^Lablist1^Gwlist1^Deflex1^
       Exlex1^(mono(lem(Word),
          ori(Dic1),
          idnum(Idnum1),
          pos(Pos),
          lab(Lablist1),
```

```
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(DefW1) )),
        DL1) ,

setof(data(Lablist2,Gwlist2,Deflex2,Exlex2,DefW2),

DefW2^Dic2^Lablist2^Gwlist2^Deflex2^Exlex2^(mo
no(lem(Word),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(DefW2) )),
       DL2) ,

  Idnum1 \= Idnum2,

  ( myorder(Idnum1,Idnum2) ->
      concat_atom([Idnum1,Idnum2],'>',IdNum);
      concat_atom([Idnum2,Idnum1],'>',IdNum)),

 % choose the appropriate separator for a given
% merge e.g. among "/ # & * @ > £" etc.

  collect(DL1,Lab1,Gw1,Def1,Ex1,DS1),
  collect(DL2,Lab2,Gw2,Def2,Ex2,DS2),

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),
  flatten(DS1,DT1),
  flatten(DS2,DT2),
  (DT1 \= DT2 -> append(DT1,DT2,DefString);
                   DefString=DT1),
  (L1 \= L2 -> (append(L1,L2,Et),
                remdup(Et,Labels));
            Labels=L1),
  (G1 \= G2 -> (append(G1,G2,Guides),
               remdup(Guides, GuideWords));
            GuideWords=G1),
  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),
  cw(def,Word,Word,D1,D2,ResDeflex,
      WeightDeflex),
  cw(ex,Word,Word,E1,E2,ResExlex,
      WeightExlex),

  sumlist([WeightLabs,WeightGws,
            WeightDeflex,WeightExlex],W),

  ( ( recorded(weights,max(Max),_),
    Max > W
    )
    ->
    true
    ;
    (recorded(weights,max(Max),Ref),
     (   W=Max -> store(mono(lem(Word),
                       ori(W),
        % we use the ori field to store the
        % the weight of the merge
                       idnum(IdNum),
                       pos(Pos),
                       lab(Labels),
                       gw(GuideWords),
                       deflex(ResDeflex),
                       exlex(ResExlex),
                       def(DefString) ))
                ;
                (   erase(Ref),
                recorda(weights,max(W),_),
                deleteref2(Max),
                store(mono(lem(Word),
                       ori(W),
                       idnum(IdNum),
                       pos(Pos),
                       lab(Labels),
                       gw(GuideWords),
                       deflex(ResDeflex),
                       exlex(ResExlex),
                       def(DefString) ))
                )

     )
    )
  ),fail.


dwmerge([W,P,W,P,w:m,m:l,noadjust],Output) :-

     setof(Entry,
          recorded(entries,Entry,_),
          Entries),
     kill_them(Entries),
     output_them(Entries,Output),!.
```

First, it finds all the information about the lemmas. It makes sure that the identification numbers are different. Then it merges all the information it found on both words and makes sure there is no duplicate. Finally, it checks whether it has already found a greater weight. If it has, it does nothing with the weight it has just found. If it has not, it deletes all the information about the previous weight and stores all the information about the new one. When this is done, it fails for two reasons: first, to be able to look for others solutions; then, so that the second clause lists the entries that produced the merged entries.

11.3. Utilities

```
output_them([],_).
output_them([Entry|MoreEntries],L) :-
        output_it(Entry,L),
        output_them(MoreEntries,L).
output_it(Entry,Stream) :- nlb(Stream),
                    writeqb(Stream,Entry),
                    writeb(Stream,'.').
kill_them([]).
```

```
kill_them([mono(lem(_),ori(_),idnum(IdNum),pos(_),
lab(_),gw(_),deflex(_),exlex(_),def(_))|MoreEntries])
                                                :-
        concat_atom([Idnum1,Idnum2],'>',IdNum),
% retrieve the 2 idnums of the merged entries
        assertz(kill(Idnum1)),
        assertz(kill(Idnum2)),
        kill_them(MoreEntries).
```

```
remdup([],[]).
remdup([X|Y],L) :-
                member(X,Y),
                remdup(Y,L),!.
```

```
remdup([X|Y],[X|Y1]) :-
                \+ member(X,Y) ,
                   remdup(Y,Y1),
                   !.
```

12. Top Three Mode

12.1. Main Predicates

```
dealwith([Word1,Pos1,Word2,Pos2,w:t3,m:l,
            noadjust],Output) :-
      dwt3([Word1,Pos1,Word2,Pos2,
       w:t3,m:l,noadjust],Output).

dwt3([Word1,Pos1,Word2,Pos2,w:t3,
        m:l,noadjust],Output) :-
   setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),
        Def^Dic^Lablist1^Gwlist1^Deflex1^Exlex1^
      (mono(lem(Word1),
          ori(Dic),
          idnum(Idnum1),
          pos(Pos1),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def) )),
         DL1) ,
```

```
       Def^Dic2^Lablist2^Gwlist2^Deflex2^Exlex2^
      (mono(lem(Word2),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos2),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def) )),
        DL2) ,
  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
```

```
    setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
   flatten(Ex1,E1),
   flatten(Ex2,E2),

   compute(lb,L1,L2,ResLabs,WeightLabs),
   compute(gw,G1,G2,ResGws,WeightGws),

cw(def,Word1,Word2,D1,D2,ResDeflex,
       WeightDeflex),

cw(ex,Word1,Word2,E1,E2,ResExlex,
       WeightExlex),
   flatten(Def2,D2),

  sumlist([WeightLabs,WeightGws,
             WeightDeflex,WeightExlex],W),

  Wneg is -W,

  recorda(top,
        Wneg-p(Word1,Idnum1,Word2,Idnum2),_),
  fail.
```

Just as in the other modes, it computes lexical proximity, but then stores the result and fails to compute the lexical proximity of all the pairs. When all the pairs have been found, the best three are displayed and all the information about the result is erased:

```
dwt3([W1,P1,W2,P2,w:t3,m:l,noadjust],Output) :-
                    findall(Pair,
                          recorded(top,Pair,_),
                         Pairs),
                     sort(Pairs,Sorted),
                     top(Sorted,Top),
                          reportraw(Top,Output),
                          reporttop(Top,Output),
                          eraseall(top),!.

dwt3([W1,P1,W2,P2,w:t3,m:l,noadjust],Output) :-
      reportnil(Output), !.
```

12.2. Utilities

```
top([Wa-p(W1a,I1a,W2a,I2a),
       Wb-p(W1b,I1b,W2b,I2b),
       Wc-p(W1c,I1c,W2c,I2c)|R],
     [Wa-p(W1a,I1a,W2a,I2a),
      Wb-p(W1b,I1b,W2b,I2b),
      Wc-p(W1c,I1c,W2c,I2c)]) :- !.

top([Wa-p(W1a,I1a,W2a,I2a),
      Wb-p(W1b,I1b,W2b,I2b)|R],
     [Wa-p(W1a,I1a,W2a,I2a),
     Wb-p(W1b,I1b,W2b,I2b)]) :- !.

top([Wa-p(W1a,I1a,W2a,I2a)|R],
       [Wa-p(W1a,I1a,W2a,I2a)]) :- !.

top([],[]).
```

```
reportnil(Out) :-
         nlb(Out),
         writeb(Out,'No match'),
         nlb(Out).

reportraw(T,Out) :-
        nlb(Out),
        writeqb(Out,T),
        writeb(Out,'.'),
        nlb(Out).

reporttop([],Out).
reporttop([Top|MoreTops],Out) :-
       reportit(Top,Out),
       reporttop(MoreTops,Out).
```

```
    reportit(Weight-p(W1,I1,W2,I2),Out) :-
       nlb(Out),
       Wneg is -Weight,
       writeb(Out,Wneg), writeb(Out,'  for the pair: '),
       nlb(Out), writeb(Out,W1),
       tabb(Out,1),writeb(Out,I1),
       tabb(Out,8),
       writeb(Out,W2),
       tabb(Out,1),
       writeb(Out,I2),
       nlb(Out),
       writeb(Out, 'i.e. the following entries : '),
       nlb(Out),
       show(I1,Out),
       show(I2,Out).
```

13. Dealing with Triplets

```
% e.g. t(wear,v,tie,n,watch,n)

% the idea is to select the Word-Idnum pairs that
% appear in more than one relation
% for instance we choose the reading for wear that
% appears in both 'wear,v tie,n' and 'wear,v
% watch,n'
% similarly we choose the reading for watch that
% appears in both 'wear,v watch,n' and 'tie,n
% watch,n'

% we give the top 3 pairings for all relations
% and then show the Idnums that satisfy the
% requirement just discussed
```

```
dealwith(t(Target,PosT,Arrow1,PosArg,Arrow2,
            PosArg),L) :-
  best3(Target,PosT,Arrow1,PosArg, Top1),
  best3(Target,PosT,Arrow2,PosArg, Top2),
  best3(Arrow1,PosArg,Arrow2,PosArg, Top3),
  nlb(L), writeb(L,Top1), nlb(L), writeb(L,Top2),
  nlb(L), writeb(L,Top3), nlb(L), nlb(L),

  select_target(Target-IdTarg,Top1,Top2,FS1),
  select_arg1(Arrow1-IdA1,Top1,Top3,FS2),
  select_arg2(Arrow2-IdA2,Top2,Top3,FS3),

  (FS1=yes -> show(IdTarg,L); true),
  (FS2=yes -> show(IdA1,L); true),
  (FS3=yes -> show(IdA2,L); true),
   !.
```

```
select_target(Lemma-Idnum,L1,L2,yes) :-
    member(W1-p(Lemma,Idnum,_,_),L1),
    member(W2-p(Lemma,Idnum,_,_),L2),!.

select_target(Lemma-Idnum,L1,L2,no).

select_arg1(Lemma-Idnum,L1,L2,yes) :-
    member(W1-p(_,_,Lemma,Idnum),L1),
    member(W2-p(Lemma,Idnum,_,_),L2), !.
```

```
select_arg1(Lemma-Idnum,L1,L2,no).

select_arg2(Lemma-Idnum,L1,L2,yes) :-
    member(W1-p(_,_,Lemma,Idnum),L1),
    member(W2-p(_,_,Lemma,Idnum),L2),!.

select_arg2(Lemma-Idnum,L1,L2,no).
```

```
best3(Word1,Pos1,Word2,Pos2,_Top) :-
  setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),
        Def^Dic^Lablist1^Gwlist1^Deflex1^Exlex1^
       (mono(lem(Word1),
         ori(Dic),
         idnum(Idnum1),
         pos(Pos1),
         lab(Lablist1),
         gw(Gwlist1),
         deflex(Deflex1),
         exlex(Exlex1),
         def(Def) )),
       DL1) ,

  setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
      Def^Dic2^Lablist2^Gwlist2^Deflex2^Exlex2^
       (mono(lem(Word2),
         ori(Dic2),
         idnum(Idnum2),
         pos(Pos2),
         lab(Lablist2),
         gw(Gwlist2),
         deflex(Deflex2),
         exlex(Exlex2),
         def(Def) )),
```

```
       DL2) ,

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),

  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),


  cw(def,Word1,Word2,D1,D2,ResDeflex,
      WeightDeflex),
  cw(ex,Word1,Word2,E1,E2,ResExlex,
      WeightExlex),

  sumlist([WeightLabs,WeightGws,WeightDeflex,
           WeightExlex],W),
```

```
        Wneg is -W,
        recorda(top,
```

```
        Wneg-p(Word1,Idnum1,Word2,Idnum2),_),
  fail.
```

```
% when all pairs have been recorded
% we can gather them all together with a findall
% clause on the recorded pairs
% and report on the results
% we end up by cleaning, i.e. erasing all recorded
% information
```

```
best3(W1,P1,W2,P2,Top) :-
                    findall(Pair,
                         recorded(top,Pair,_),
                        Pairs),
                    sort(Pairs,Sorted),
                    top(Sorted,Top),
                    eraseall(top),!.
```

14. Computing Lexdis Weight

This mode amounts to launching the query [Word,Pos,Word,P,m:g]:

```
% to compute the LEXDIS weight of all semdic
% entries, execute the query : w(W,P).

dealwith(w(Word,Pos),Output) :-

   setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),

Def^Dic^Idnum^Lablist1^Gwlist1^Deflex1^Exlex1^
       (mono(lem(Word),
          ori(Dic),
          idnum(Idnum),
          pos(Pos),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def)   )),
        DL1),

   setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
       Def^Dic2^Idnum2^Lablist2^Gwlist2^Deflex2^
       Exlex2^(mono(lem(Word),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def)   )),
       DL2),

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

% we distribute the data just collected : labels,
% guide words,  definition core items, example core
% items
```

```
  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),
  flatten(Ex1,E1),
  flatten(Ex2,E2),

  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),

% computing the weight to be assigned to both
% labels and guide words


  cw(def,Word,Word,D1,D2,ResDeflex,
        WeightDeflex),
  cw(ex,Word,Word,E1,E2,ResExlex,
        WeightExlex),

% idem for definition core items and example
% core items

  metameet(Word,Word,Metaweight),
  rogetmeet(Word,Word,Rogetweight),
  indicmeet(Word,Pos,Word,Pos,Indicweight),
  collmeet(Word,Pos,Word,Pos,Collweight),
  envirmeet(Word,Word,Envirweight),
  wordnetmeet(Word,Word,Pos,WordNetweight),

  % weight assignation to collocate field sharing,
% Roget's category sharing,
  % indicator sharing, collocate sharing, sharing of
% environment

  sumlist([WeightLabs,WeightGws,WeightDeflex,
           WeightExlex,Metaweight,Rogetweight,
           Indicweight,Collweight,Envirweight,
```

```
            WordNetweight],GW),

% writing doc file
% swritef(Towrite,'%60l;%15l;%15l\n',
 %              [Word,Pos,GW]),
% writeb(Output,Towrite).

% writing weights file
   Entry= w(Word,Pos,GW),
   output_it(Entry,Output).
```

15. Friend Mode

This mode takes a word as a pivot and a minimum weight as a threshold. The results are all the words which equal or exceed the threshold in combination with the pivot:

```
dealwith(friends(Word1,Pos1,w:MinimumWeight),
              Output) :-
   setof(data(Lablist1,Gwlist1,Deflex1,Exlex1),
        Def^Dic^Idnum^Lablist1^Gwlist1^
       Deflex1^Exlex1^(mono(lem(Word1),
          ori(Dic),
          idnum(Idnum),
          pos(Pos1),
          lab(Lablist1),
          gw(Gwlist1),
          deflex(Deflex1),
          exlex(Exlex1),
          def(Def)   )),
         DL1),

   setof(data(Lablist2,Gwlist2,Deflex2,Exlex2),
       Def^Dic2^Idnum2^Lablist2^Gwlist2^
      Deflex2^Exlex2^(mono(lem(Word2),
          ori(Dic2),
          idnum(Idnum2),
          pos(Pos2),
          lab(Lablist2),
          gw(Gwlist2),
          deflex(Deflex2),
          exlex(Exlex2),
          def(Def)   )),
       DL2),

  collect(DL1,Lab1,Gw1,Def1,Ex1),
  collect(DL2,Lab2,Gw2,Def2,Ex2),

% we distribute the data just collected : labels,
% guide words, definition core items, example core
% items

  flatten(Lab1,L1),
  flatten(Lab2,L2),
  flatten(Gw1,G1),
  flatten(Gw2,G2),
  flatten(Def1,D1),
  flatten(Def2,D2),

  cw(def,Word1,Word2,D1,D2,ResDeflex,
       WeightDeflex),
  cw(ex,Word1,Word2,E1,E2,ResExlex,
       WeightExlex),

  flatten(Ex1,E1),
  flatten(Ex2,E2),

  compute(lb,L1,L2,ResLabs,WeightLabs),
  compute(gw,G1,G2,ResGws,WeightGws),

  metameet(Word1,Word2,Metaweight),
  rogetmeet(Word1,Word2,Rogetweight),
  indicmeet(Word1,Pos1,Word2,Pos2,Indicweight),
  collmeet(Word1,Pos1,Word2,Pos2,Collweight),
  envirmeet(Word1,Word2,Envirweight),

  (Pos1=Pos2 ->
              wordnetmeet(Word1,Word2,Pos1,
                        WordNetweight);
              WordNetweight=0),

  % weight assignation to collocate field sharing,
% Roget's category sharing,
  % indicator sharing, collocate sharing, sharing of
% environment

  sumlist([WeightLabs,WeightGws,WeightDeflex,
           WeightExlex, Metaweight,
            Rogetweight,Indicweight,Collweight,
           Envirweight,WordNetweight],GW),


  % adjusting weights according to lexical weight:
% remmed or remmable
  % if no threshold is specified, neg weights also
% recorded

  (MinimumWeight=none -> true;
              GW >= MinimumWeight),

   Entry= f(Word2,Pos2,GW),
   output_it(Entry,Output).
```

Chapter VI: Evaluation

0. Introduction

In this chapter, I analyze the results that *Lexdis* produces on different tests. Measuring lexical proximity is not an exact science. The tests I performed are just probes. There are way too few of them to be statistically representative.

In section 1, I describe the algorithm of the original version of *Lexdis* as well as the algorithm of the other versions I used to try to improve the weighting system. In section 2, I present four tests which contrast *Lexdis's* and my judgement of lexical proximity on different types of pair. In section 3, I use *Lexdis* as a WSD tool in combination with collocations. In section 4, I test the triplet mode.

1. Algorithms

1.0. Introduction

This chapter presents the different versions of *Lexdis* I worked with as well as the algorithm they use. In all these versions, I altered a number of parameters which influenced the results. The parameters are:

- *semdic or lightdic*: *lightdic* is a version of *semdic* which does not encompass WordNet glosses. It makes it possible to assess clearly whether the glosses improve the programme or not[11];
- *Lexdis* or *Lexdiswn*: *Lexdiswn* is a version of *Lexdis* which uses WordNet's semantic relationships (cf. Chapter 4.8);
- *weighting*: each version emphasizes a particular type of information (e.g. definitions, collocations, etc.) by increasing the original weight.

The rest of this section describes all the versions I used to run the tests. The first version is the reference version. The definitive version, i.e. the one I describe in the previous chapter, is version 7. Note that the difference between a full version (e.g. version 1) and a "b" version (e.g. version 1b) is that the second version uses *lightdic* instead of *semdic*.

[11] It is also interesting to note that lightdic loads twice as fast as semdic (about 10 seconds vs. 20 seconds). Once loaded, the results are provided in the same amount of time.

1.1. Version 1: *Lexdis*

This is the original version of *Lexdis*. Its parameters are:

- *Labels*: each shared label is worth **4**.
- *Guide words*: each shared guide word is worth **12**.
- *Definitions*: if the first word is used in the definition of the second word and vice versa, there is a bonus of **20**. Also, the **number** of words the definitions have in common is weighted. If it does not exceed a threshold of **2 words**, it is multiplied only by **2** as the sharing may be incidental. If it does, it is multiplied by **3**. The result is this number plus the bonus.
- *Examples*: if the first word is used as an example for the second word and vice versa, there is a bonus of **5**. The bonus is lower than that of definitions because it is less reliable. There is no incidental threshold. The **number** of shared words added to the bonus is the result.
- *Co-occurrence of collocations (metameet):* the **number** of co-occurrences of the second word in alphabetical order is divided by **2**. If this word is "person" or "object", it is divided by **16** because of their poor semantic discriminatory power.
- *Roget's Thesaurus* has 3 hierarchical levels. Sharing the broadest category is worth **1**; sharing the first and the second category is worth **2**; sharing all three is worth **3**.
- *Indicators:* if the first word is a member of the indicator list of the second or vice versa, there is a bonus of **5**. There is also an incidental threshold set to **2 words** regarding the **number** of shared indicators. It is multiplied by **2** if it does not exceed the threshold. If it does, it is multiplied by **4**. The result is this number plus the bonus.
- *Collocations:* if the first word is a member of the collocation list of the second or vice versa, there is a bonus of **5**. There is also an incidental threshold set to **2 words** regarding the **number** of shared collocates. It is multiplied by **2** if it does not exceed the threshold. If it does, it is multiplied by **4**. The result is this number plus the bonus.
- *Environment:* if the first word is a member of the environment list of the second or vice versa, there is a bonus of **3**. Also, if the two lists share only **1** word, it is not taken into account. If they share more, the result is the number of shared words plus the bonus.

1.2. Version 2: *Lexdiswn*

It uses the same parameters as version 1, plus WordNet's semantic relationships.

1.3. Version 3: Lexdis

This version emphasizes the importance of definitions and examples.

- *Definitions*: if the first word is used in the definition of the second or vice versa, the bonus goes up to **40**. If the number of shared words exceeds the threshold of the **2 words**, it is multiplied by **5**.
- *Examples*: if the first word is used as an example for the second word and vice versa, the bonus goes up to **10**. Also, the number of shared words is multiplied by **2**.

1.4. Version 4: Lexdiswn

This is the same version as version 3, except that it uses WordNet's semantics relationships.

1.5. Version 5: Lexdis

This version emphasizes the importance of collocations:

- *Co-occurrences of collocations*: the **number** of co-occurrences is kept as such (i.e. it is not divided by **2**).
- *Collocations*: if the first word is a collocation of the second word and vice versa, there is a bonus of **10**. The minimal incidental threshold is still set to **2 words**. If it does not exceed that threshold, it is multiplied by **3** and if it does, it is multiplied by **6**.

1.6. Version 6: Lexdiswn

This is the same version as version 6, except that it uses WordNet's semantic relationships.

1.7. Version 7: Lexdiswn

This is my final version. The algorithms take into account the results from section 2. This version emphasizes the importance of definitions and collocations:

- It uses *semdic*.
- It uses WordNet's semantic relationships.
- *Definitions*: the minimal threshold is still of **2 words**, but if the **number** of shared words exceeds it, it is multiplied by **5** (instead of 3).
- *Co-occurrence of collocations*: the **number** of co-occurrences is kept as such (i.e. it is not divided by **2**).

- *Collocations*: the bonus if the word appears in the collocations list of the other word (and vice versa) is **20** (instead of 10). The minimal threshold is still **2 words**, but if the **number** of shared collocations exceeds it, it is multiplied by **5**.

2. Heuristic Tests

2.0. Introduction

In this section, I test the different versions of *Lexdis* listed in section 1 on different kinds of pair. The first features a concrete word as a pivot while the second features an abstract word. The third test features a verb with its collocations. The last test features different kinds of pair whose lexical proximity I assess as very low. Note that a couple of protocols (i.e. what appears on the screen while the programme runs) are available in the appendix.

2.1. The Pivot is a Concrete Word

In this test, I computed lexical distance between "radiator" and other words:

#	Queries	Versions of *Lexdis*												
		1	1b	2	2b	3	3b	4	4b	5	5b	6	6b	**7**
1	radiator - heat	50	35	60	45	86	60	96	70	50	50	60	45	**72**
2	radiator - warmth	4	2	4	2	6	3	6	3	4	4	4	2	**4**
3	radiator - water	38	29	40	31	72	56	74	58	38	38	40	31	**48**
4	radiator - cold	6	6	6	6	10	10	10	10	6	6	6	6	**6**
5	radiator - car	25	24	31	30	41	39	47	45	25	25	31	30	**41**
6	radiator - door	18	15	23	20	28	23	33	28	18	18	20	20	**31**
7	radiator - mouse	3	3	8	8	5	5	10	10	3	3	8	8	**8**
8	radiator - computer	2	2	7	7	3	3	8	8	2	2	7	7	**7**
9	radiator - literature	1	1	1	1	2	2	2	2	1	1	1	1	**1**
	mean:	16	13	20	17	28	22	32	26	16	16	20	17	**24**

The order in which the queries are listed reflects my feeling of lexical proximity: the higher, the closer. First of all, it seems to me that the results are more representative when semdic is used instead of lightdic ("b" versions) even though this does not affect the general order between the queries. Interestingly, the pair "radiator – warmth" produces a low number. This is due to the fact that a radiator fights against heat, as is the case in a car. Since "warmth" has a positive connotation (think about "it's nice and warm in here"), it is not generally associated with a

radiator. “Heat”, on the other hand, has a more negative connotation. Generally speaking, WordNet’s relationships (versions 2, 2b, 4, 4b, 6, 6b) do not harm, quite the reverse.

2.2. The Pivot is an Abstract Word

This test is similar to the previous one, expect for the fact that the pivot is an abstract word.

#	Queries	*Versions of Lexdis*												
		1	1b	2	2b	3	3b	4	4b	5	5b	6	6b	**7**
1	literature - history	41	34	47	40	70	58	76	64	43	43	49	42	**69**
2	literature - book	22	18	28	24	38	31	44	37	22	22	28	24	**40**
3	literature - war	3	3	3	3	6	6	6	6	3	3	3	3	**3**
4	literature - love	13	10	13	10	25	19	25	19	13	13	13	10	**13**
5	literature - pen	19	10	19	10	30	14	30	14	19	19	19	10	**27**
6	literature - chess	0	0	0	0	0	0	0	0	0	0	0	0	**0**
7	literature - door	4	4	4	4	8	8	8	8	4	4	4	4	**4**
8	literature - mouse	1	1	1	1	2	2	2	2	1	1	1	1	**1**
	mean:	13	10	14	12	22	17	24	19	13	13	15	12	**20**

Of course, there exists literature on almost everything. I am surprised to see #3 with such a small number, but the rest seems representative. #5 deserves its third position and if #3 has such a low number, then the result of #4 is not that surprising. Still, #3 should be higher than #7, but their lexical proximity is only the result of shared words in examples, which is far from the most reliable semantic axis. Finally, WordNet’s versions (uneven numbers) back up the original versions (even numbers).

2.3. A Verb and its Collocations

The aim of this test is to make sure that *Lexdis* gives a high value to collocations (**cf. table 1, p.73**). Pairs 1a to 1f should give high results as the nouns are all subject collocations. This is this test that influenced the final version of *Lexdis* which emphasizes the contribution of collocations. Indeed, the mean of version 7 is twice as high as that of version 1, a much higher discrepancy compared to the means of the two previous tests. I included two words that are not collocations (i.e. “history” and “chess”) to point out the difference between a high and a low result.

#	Queries	*Versions of Lexdis*												
		1	1b	2	2b	3	3b	4	4b	5	5b	6	6b	**7**
1a	go off - alarm	31	22	31	22	51	36	51	36	36	36	36	27	**58**
1b	go off - gun	24	15	2	15	39	22	39	22	29	29	29	20	**47**
1c	go off - bomb	18	11	18	11	28	15	28	15	23	23	23	16	**39**
1d	go off - milk	19	18	19	18	31	29	31	29	24	24	24	23	**34**
1e	go off - meat	15	15	15	15	19	19	19	19	20	20	20	20	**30**
1f	go off - clock	14	12	14	12	21	18	21	18	19	19	19	17	**29**
7	go off - siren	11	11	11	11	19	19	19	19	11	11	11	11	**17**
8	go off - history	7	9	7	9	17	17	17	17	9	9	9	9	**9**
9	go off - chess	0	0	0	0	0	0	0	0	0	0	0	0	**0**
	mean:	15	13	14	13	25	19	25	19	19	19	16	16	**29**

Table 1

2.4. Unrelated Words

I include in this test pairs which to me have (almost) nothing in common:

#	Queries	**Versions of Lexdis**												
		1	1b	2	2b	3	3b	4	4b	5	5b	6	6b	**7**
1a	orange - bed	1	1	1	1	2	2	2	2	1	1	1	1	**1**
1b	car - sofa	4	4	4	4	6	6	6	6	5	5	5	5	**5**
1c	grass - cupboard	3	1	3	3	5	2	5	2	3	3	3	1	**3**
1d	repair - mathematics	0	0	0	0	0	0	0	0	0	0	0	0	**0**
1e	uncompromising - garbage	4	4	4	4	6	6	6	6	4	4	4	4	**4**
1f	eat - planet	0	0	0	0	0	0	0	0	0	0	0	0	**0**
1g	shoe - school	3	3	3	3	6	6	7	7	4	4	5	5	**5**
	mean:	2	2	2	2	4	3	4	3	2	2	3	2	**3**

All the results are low, which is a good thing. Note that it is not the final version that gives the highest numbers.

2.5. Conclusion

The results of these tests seem to indicate that *Lexdis* rarely gives wrong results. The scale of the final version is:

- from 0 to10: unrelated;
- from 10 to 20: there is a not insignificant link;
- from 20 to 30: there is a strong link;
- from 30 to ~: there is an undeniable link.

Not only does this range work for almost all the words – I would exclude #3 from test 2 ("literature – war") which deserves more than 3 – but the order in which *Lexdis* classifies the pairs globally corresponds to my own classifications.

3. Collocations Test

Lexdis can also be used in combination with collocate lists. In this test, I reproduce a test carried out by Michiels on the verb "to go through"[12]. The collocations associated with their word sense come from Oxford Dictionary of Current Idiomatic English:

(1) **to consume**: stock, store, food, beer, fortune;
(2) **to search**: room, pocket, paper;
(3) **to perform**: marriage, initiation, matriculation, ceremony;
(4) **to rehearse**: fact, argument, scene, text;
(5) **to be published**: book, title, article, printing, edition;
(6) **to experience**: operation, pain, ordeal, fire.

The test sentences are:

- He thought that she had gone through the **bins.**
- The inspector wanted the teachers to go through his **report**.
- She was expected to go through a daily **ritual**.
- We know the **suffering** you went through.
- It is the **training** the students will be expected to go through.
- You don't know the **torture** I have gone through.
- I think that they went through five **bottles**.
- The report went through five **editions**.

Lexdis computes the lexical distance of all the pairs. The results are to be found in **table 2, p.75**.

First, a note on the conventions:

- the collocations in bold are those which produced the highest number;
- the framed means with a grey background are the means that *Lexdis* got right;
- the framed means without a grey background are incorrect results;

[12] In Michiels's test, *Lexdis* works with a parser to choose the best word sense.

	Lexdiswn Version 7							
Queries	bin	report	ritual	suffering	training	torture	bottle	edition
stock	6	**41**	**5**	2	26	17	29	**25**
store	**43**	9	3	2	**28**	1	21	4
food	0	9	2	4	26	**20**	9	4
beer	15	2	0	1	0	0	**67**	1
fortune	3	15	2	**9**	20	6	4	2
(1) mean:	13	15	2	4	20	9	**26**	7
room	29	6	5	2	7	10	**30**	8
pocket	**30**	7	5	1	23	2	13	11
paper	9	**188**	**6**	**4**	**23**	**22**	6	**56**
(2) mean:	**23**	67	5	2	18	11	16	25
marriage	1	6	12	**5**	1	1	1	1
initiation	1	14	16	1	**32**	**4**	**3**	2
matriculation	0	0	0	0	0	0	0	0
ceremony	**3**	**19**	**66**	1	29	0	**3**	**4**
(3) mean:	1	10	**24**	2	16	1	2	2
fact	2	43	0	1	2	3	2	2
argument	3	32	2	3	15	7	1	17
scene	2	39	**23**	**7**	**29**	**27**	**5**	38
text	**4**	**45**	5	3	5	4	0	**72**
(4) mean:	3	**40**	8	4	13	10	2	32
book	**13**	46	5	**3**	18	**11**	**5**	84
title	6	29	0	2	26	5	1	9
article	3	**66**	1	0	**41**	2	3	19
printing	2	7	0	0	4	1	1	57
edition	1	42	**6**	0	19	1	1	**273**
(5) mean:	5	38	2	1	22	4	2	**88**
operation	**3**	12	**41**	19	**39**	18	2	**19**
pain	0	2	0	**115**	7	**94**	**8**	1
ordeal	0	4	3	4	5	16	2	1
fire	**3**	**22**	4	12	11	31	1	5
(6) mean:	2	10	12	**38**	**16**	**40**	3	7

Table 2

- the unframed means with a grey background are those that *Lexdis* should have found.

At first sight, the results are very encouraging. *Lexdis* manages to induce the correct word sense of 6 sentences out of 8 (i.e. 75%) if we take the mean as a point of reference. Since we use collocate lists, it seems risky to use only the highest value of the collocate list.

The worst case is the result of the second test sentence. Not only does *Lexdis* find the wrong result, there is another word sense close to the intended one. But if we have a closer look at it, it is not that bad. First, the best and incorrect match produces such a high number because the pair "paper" and "report" produces a high number, which is logical. Note that the two other collocations – "room" and "pocket" – produce low numbers. In other words, high standard deviation[13] may point out that the result is due to chance. Indeed, the results produced by the collocations of the correct word sense (#4) do not differ a lot from the standard deviation: 43, 32, 39 and 45 for a mean of 40. Word sense #5 is also very close, but it is due to the high proximity of "article" and "report". Here too, standard deviation is higher than for word sense #4. On the other hand, the failure of test sentence #5 is irreversible: "training" happens to produce high numbers with a lot of different collocates. As a very abstract and general word, the lexical information available is not discriminatory enough and also, training is a fine grained word.

The other results speak for themselves. *Lexdis* finds the correct word sense without any ambiguity and for the good reasons: the collocations of the list have a higher lexical proximity than that of the other lists.

It must be noted that the results are based on very low number of collocations. It seems to me that 4 collocations for a word sense is a minimum to avoid getting chance results. The problem is that collocations are based on frequency, but if a word sense has only 3 collocations, there is nothing to do about it. It would also be best if all the word senses had the same number of collocations, but that too is not up to the user.

If *Lexdis*'s results are not fully satisfactory, there may be two reasons. First, *Lexdis* may not work properly, or its algorithms may not suit the task. Another possibility is that the collocations and word sense inventory should be revised. For instance, the previous distinction between "perform" and "rehearse" may be too fine or too context-dependent so that *Lexdis* succeeds in finding the intended word sense. This encouraged Michiels to adapt the word senses of "to go through" and their collocations:

(1) **to consume**: money, food, drink;

(2) **to search**: room, pocket, clothes, cupboard, wardrobe, luggage, suitcase, trunk;

[13] "the amount by which a single measurement differs from a fixed value such as the mean" (COED)

(3) **to perform, to rehearse**: marriage, initiation, scene, lesson, programme, ceremony, formality, procedure;

(4) **to examine**: fact, argument, subject, file, mail, text, list, document;

(5) **to be published**: book, title, article, printing, edition;

(6) **to experience**: operation, pain, ordeal, apprenticeship, fire, phase, stage, process, experience, experiment.

The test sentences remain the same and as well as the conventions for the table: (**cf. table 3a and 3b**). This time, *Lexdis* gets 7 sentences out of 8 (87%). When the result is correct, it is not ambiguous. The only case where *Lexdis* does not find the intended word is test sentence 5. First, the results are very close: 23 (*Lexdis*'s result) vs. 22 (intended word sense). If we look at the collocation list of the word senses in question, it is far from surprising that we get a tie. The word senses are "to perform / to rehearse" and "to experience" and the word from the test sentence is "training". I would have a small preference for the second word sense, but both are possible, which is confirmed by *Lexdis*.

	Means from Table 3b							
(1) mean:	2	11	2	2	17	8	**37**	4
(2) mean:	**16**	2	1	1	7	2	17	4
(3) mean:	2	23	**29**	2	**23**	7	2	16
(4) mean:	2	**61**	1	1	1	4	2	7
(5) mean:	5	38	2	1	18	4	2	**88**
(6) mean:	2	10	10	**22**	22	**21**	4	6

Table 3a

	Lexdiswn version 7							
Queries	bin	report	ritual	suffering	training	torture	bottle	edition
money	1	**24**	1	1	22	1	2	**5**
food	0	9	**2**	**4**	**26**	**20**	9	4
drink	**5**	1	**2**	2	2	3	**100**	2
(1) mean:	2	11	2	2	17	8	37	4

Queries	bin	report	ritual	suffering	training	torture	bottle	edition
room	29	6	**5**	**2**	7	**10**	30	8
pocket	**30**	**7**	**5**	1	**23**	2	13	**11**
clothes	1	2	0	**2**	4	2	3	0
cupboard	22	0	1	0	3	0	5	1
wardrobe	22	1	0	0	4	1	4	9
luggage	6	0	0	0	0	1	9	0
suitcase	4	0	0	0	0	1	18	0
trunk	16	2	0	**2**	16	2	**50**	1
(2) mean:	**16**	2	1	1	7	2	17	4
marriage	1	6	12	5	1	1	1	1
initiation	1	14	19	1	32	4	3	2
scene	2	39	23	**7**	29	**27**	**5**	38
lesson	2	**51**	20	3	25	16	2	5
programme	0	49	28	2	24	5	4	**70**
ceremony	**3**	19	**66**	1	29	0	3	4
formality	2	2	5	0	6	1	0	2
procedure	1	7	56	0	**38**	4	1	5
(3) mean:	2	23	**29**	2	23	7	2	16
fact	2	43	0	1	2	3	2	2
argument	3	32	2	3	15	7	1	17
subject	7	72	4	**18**	**24**	**8**	5	45
file	**30**	23	1	1	1	5	**40**	6
mail	13	20	**6**	1	5	2	11	13
text	4	45	5	3	5	4	0	**72**
list	4	25	1	0	0	2	2	6
document	1	**78**	1	0	0	4	2	12
(4) mean:	2	**61**	1	1	1	4	2	7
book	**13**	46	5	**3**	18	**11**	**5**	84
title	6	29	0	2	26	5	1	9
article	3	**66**	1	0	**41**	2	3	19
printing	2	7	0	0	4	1	1	57
edition	1	42	**6**	0	1	1	1	**273**
(5) mean:	5	38	2	1	18	4	2	88

Queries	bin	report	ritual	suffering	training	torture	bottle	edition
operation	3	12	**41**	19	39	18	2	19
pain	0	2	0	**115**	7	**94**	**8**	1
ordeal	0	4	3	4	5	16	2	1
apprenticeship	0	0	4	3	15	1	0	0
fire	3	**22**	4	12	11	31	1	5
phase	3	7	16	17	17	16	7	**20**
stage	**5**	20	5	3	19	1	5	7
process	0	21	22	2	53	21	5	8
experience	1	7	4	41	**54**	8	2	0
experiment	2	2	5	1	3	2	4	2
(6) mean:	2	10	10	22	22	21	4	6

Table 3b

4. Triplet Mode

The triplet mode makes it possible to use *Lexdis* as a WSD tool. In this test, I investigate the verb "to eat". As explained in chapter 3.6, a word sense is considered as correct if it is part of the result of the two queries. Note that all the original sentences come from the British National Corpus. The protocol of this test is to be found in the appendix.

Query	Right	Wrong
eat	X	
seed	X	
root	X	
(1)	**100%**	
eat		A
salad	X	
bread	X	
(2)	**67%**	
eat	X	
ham	X	
bean	X	
(3)	**100%**	
eat	X	
meat		A
pie	X	
(4)	**67%**	
eat		A
pear	X	
peach	X	
(5)	**67%**	

"A" in the "wrong" column means that *Lexdis* cannot find a satisfactory word sense. When *Lexdis* cannot find the word sense of "to eat", it seems to me that this is due to the quality of the lexical information. For instance, the following word senses of "to eat" are considered as distinct:

- when you eat something or when you eat you put food into your mouth chew it and swallow it, to put or take food into the mouth chew it and swallow it;
- to take in through the mouth and swallow solid food or soup.

The test seems to indicate that these word senses would be good candidates for a merge. This is also the case for "meat" (#4):

- the flesh of an animal when it is used for food, meat is flesh taken from an animal that has been killed so that people can cook it and eat it;
- the flesh of animals apart from fish and birds which is eaten.

It seems that the triplet mode provides very interesting results which emphasize the inconvenient of working with different lexical sources. It seems that there is still work to be done as far as merging the databases is concerned. The problem is that this can only be done computationally while making sure that correct word senses are merged.

5. Conclusion

Lexdis is a very promising application. It displays consistent results both on heuristic and empirical tests and on the collocation test. When a result is wrong or unexpected, it is not due to *Lexdis's* algorithms being wrong, but to the lack of discriminatory information or to chance. *Lexdis* can also be used to question word sense distinction. The test of the triplet mode points out that it is possible to improve the quality of the lexical databases, which is far from an easy task.

Appendix

1. Pivot as a Concrete Word

QUERY: [radiator, n, warmth, n, m:g]

radiator with POS=n is related to warmth with POS=n with weight=4 as follows:
Shared words in definition: [heat, energy] -> weight: 4

QUERY: [radiator, n, heat, n, m:g]

radiator with POS=n is related to heat with POS=n with weight=72 as follows:
Shared words in definition: [hot, heat, transferred, energy, heating, system] -> weight: 50
Shared words in examples: [water, heating, room, warm] -> weight: 4
Cooccurrence in R/C-Oxf/Hach indic db -> weight: 5
Cooccurrence in R/C-Oxf/Hach extended lemma db -> weight: 3
Sharing WordNet path -> weight: 10

QUERY: [radiator, n, cold, n, m:g]

radiator with POS=n is related to cold with POS=n with weight=6 as follows:
Shared words in definition: [hot, heat] -> weight: 4
Shared words in examples: [water, warm] -> weight: 2

QUERY: [radiator, n, mouse, n, m:g]

radiator with POS=n is related to mouse with POS=n with weight=8 as follows:
Shared words in definition: [device] -> weight: 2
Shared words in examples: [room] -> weight: 1
Sharing WordNet path -> weight: 5

QUERY: [radiator, n, door, n, m:g]

radiator with POS=n is related to door with POS=n with weight=31 as follows:
Shared Labels: [hh] -> weight: 4
Shared words in definition: [metal, vehicle, object, room] -> weight: 20
Shared words in examples: [room, car] -> weight: 2
Sharing WordNet path -> weight: 5

QUERY: [radiator, n, literature, n, m:g]

radiator with POS=n is related to literature with POS=n with weight=1 as follows:
Shared words in examples: [car] -> weight: 1

QUERY: [radiator, n, car, n, m:g]

radiator with POS=n is related to car with POS=n with weight=41 as follows:
Shared Labels: [au] -> weight: 4
Shared words in definition: [vehicle, engine, motor, purpose, room] -> weight: 25
Shared words in examples: [car] -> weight: 6
Sharing WordNet path -> weight: 6

QUERY: [radiator, n, water, n, m:g]

radiator with POS=n is related to water with POS=n with weight=48 as follows:
Shared words in definition: [contains, water, pipes, room] -> weight: 40
Shared words in examples: [water] -> weight: 6
Sharing WordNet path -> weight: 2

QUERY: [radiator, n, computer, n, m:g]

radiator with POS=n is related to computer with POS=n with weight=7 as follows:
Shared words in definition: [electric] -> weight: 2
Sharing WordNet path -> weight: 5

QUERY: nadamas

cputime : 2.09041

2. A Verb and its Collocations

QUERY: [go off, v, milk, n, m:g]

go off with POS=v is related to milk with POS=n with weight=34 as follows:
Shared words in definition: [food, drink] -> weight: 4
Shared words in examples: [best, baby, bottle, coffee, tea] -> weight: 10
Cooccurrence in R/C-Oxf/Hach collocates db -> weight: 20

QUERY: [go off, v, meat, n, m:g]

go off with POS=v is related to meat with POS=n with weight=30 as follows:
Shared Labels: [fo] -> weight: 4
Shared words in definition: [food, drink] -> weight: 4
Shared words in examples: [think, left] -> weight: 2
Cooccurrence in R/C-Oxf/Hach collocates db -> weight: 20

QUERY: [go off, v, alarm, n, m:g]

go off with POS=v is related to alarm with POS=n with weight=58 as follows:
Shared words in definition: [happen, sudden, loud, noise, event, sound] -> weight: 30
Shared words in examples: [new, night, give] -> weight: 8

Cooccurrence in R/C-Oxf/Hach collocates db -> weight: 20

QUERY: [go off, v, clock, n, m:g]

go off with POS=v is related to clock with POS=n with weight=29 as follows:
Shared words in definition: [happen, leave] -> weight: 4
Shared words in examples: [think, new, heating, work, give] -> weight: 5
Cooccurrence in R/C-Oxf/Hach collocates db -> weight: 20

QUERY: [go off, v, bomb, n, m:g]

go off with POS=v is related to bomb with POS=n with weight=39 as follows:
Shared words in definition: [explodes, event, explode] -> weight: 15
Shared words in examples: [new, night, bombs, meeting] -> weight: 4
Cooccurrence in R/C-Oxf/Hach collocates db -> weight: 20

QUERY: [go off, v, siren, n, m:g]

go off with POS=v is related to siren with POS=n with weight=17 as follows:
Shared words in definition: [loud, noise, sound] -> weight: 15
Shared words in examples: [night, hear] -> weight: 2

QUERY: [go off, v, gun, n, m:g]

go off with POS=v is related to gun with POS=n with weight=47 as follows:
Shared words in definition: [discharge, sudden, loud, noise] -> weight: 20
Shared words in examples: [hear, gun] -> weight: 7
Cooccurrence in R/C-Oxf/Hach collocates db -> weight: 20

QUERY: [go off, v, chess, n, m:g]

go off with POS=v is ***not*** related to chess with POS=n (Adjusted Weight = 0).

QUERY: [go off, v, history, n, m:g]

go off with POS=v is related to history with POS=n with weight=9 as follows:
Shared words in definition: [event] -> weight: 2
Shared words in examples: [minutes, new, city, book, work, look, light] -> weight: 7

QUERY: nadamas

cputime : 1.12321

3. Triplet Mode

QUERY: t(eat, seed, root)

[-2-p(eat, ci21841/co18049, seed, ci65769), -2-p(eat, ci21841/co18049, seed, seed%1:20:02::), -2-p(eat, eat%2:34:00::, seed, ci65769)]
[-2-p(eat, ci21841/co18049, root, ci63140), -2-p(eat, ci21841/co18049, root, lg46947), -2-p(eat, eat%2:30:00::, root, ci63140)]
[-6-p(seed, lg48272, root, lg46947), -5-p(seed, co51596, root, lg46947), -4-p(seed, ci65769, root, ci63140)]

eat v [consume, food] [] ci21841/co18049 [chinese, food, tired, vegetarian, meat, cold, feel, eating, oclock]
Def: [when you eat something or when you eat you put food into your mouth chew it and swallow it, to put or take food into the mouth chew it and swallow it]

seed n [] [] ci65769 [hot, weather, lettuces, suddenly, ran, home, holiday, onions, retired]
Def: if a plant esp one which is grown for food goesruns to seed it produces flowers and seeds because it has not been picked early enough

root n [part] [] ci63140 [girl, carefully, pulled, weed, roots, desert, lived, berries, irises, shallow]
Def: the part of a plant which grows down into the earth to obtain water and food and which holds the plant firm in the ground

QUERY: t(eat, salad, bread)

[-6-p(eat, lg63381, salad, lg47417/lg47420), -6-p(eat, lg63383, salad, lg47417/lg47420), -6-p(eat, lg63385, salad, lg47417/lg47420)]
[-2-p(eat, ci21841/co18049, bread, bread%1:13:00::), -2-p(eat, ci21841/co18049, bread, ci8062/co6619), -2-p(eat, ci21841/co18049, bread, co6622)]
[-35-p(salad, lg47417/lg47420, bread, ci8062/co6619), -24-p(salad, lg47417/lg47420, bread, bread%1:13:00::), -22-p(salad, lg47417/lg47420, bread, bread%1:21:00::)]

salad n [] [fo] lg47417/lg47420 [green, lettuce]
Def: [ame a soft mixture mainly of small pieces of a stated food as in chickentunaegg salad served cold and often between pieces of bread, a mixture of foods usu mainly vegetables served cold and sometimes esp when other foods are added as in a chickencheese salad as the main dish at a meal]

bread n [] [] bread%1:13:00:: []
Def: food made from dough of flour or meal and usually raised with yeast or baking powder and then baked

QUERY: t(eat, ham, bean)

[-6-p(eat, lg63381, ham, lg31977), -6-p(eat, lg63383, ham, lg31977), -4-p(eat, lg63385, ham, lg31977)]
[-6-p(eat, lg63381, bean, lg17879), -6-p(eat, lg63381, bean, lg17881), -6-p(eat, lg63381, bean, lg17882)]
[-6-p(ham, lg31977, bean, lg17879), -6-p(ham, lg31977, bean, lg17881), -6-p(ham, lg31977, bean, lg17882)]

eat v [] [fo] lg63381 [dinner, big, house, money]
Def: to take in through the mouth and swallow solid food or soup

ham n [] [fo] lg31977 []
Def: preserved meat from a pigs leg considered as food

bean n [] [fo] lg17879 []
Def: a seed of any of various upright climbing plants esp one that can be used as food

QUERY: t(eat, meat, pie)

[-44-p(eat, ci21841/co18049, meat, ci45185/co36007), -35-p(eat, eat%2:34:02::, meat, ci45185/co36007), -32-p(eat, lg63383, meat, ci45185/co36007)]
[-5-p(eat, lg63383, pie, lg42144), -4-p(eat, lg63381, pie, lg42144), -4-p(eat, lg63385, pie, lg42144)]
[-29-p(meat, lg37423, pie, lg42144), -29-p(meat, lg37425, pie, lg42144), -29-p(meat, meat%1:20:00::, pie, lg42144)]

eat v [] [fo, md] lg63383 [tigers, meat, horses]
Def: to use regularly as food

pie n [] [fo] lg42144 [apple, meat]
Def: often in comb an often round pastry case filled with meat or fruit baked usu in a deep dish pie dish

QUERY: t(eat, pear, peach)

[0-p(eat, ci21841/co18049, pear, ci53510/co42000), 0-p(eat, ci21841/co18049, pear, lg41477), 0-p(eat, ci21841/co18049, pear, pear%1:13:00::)]
[-4-p(eat, lg63381, peach, ci53479/co41977#lg41444), -4-p(eat, lg63383, peach, ci53479/co41977#lg41444), -4-p(eat, lg63385, peach, ci53479/co41977#lg41444)]
[-41-p(pear, ci53510/co42000, peach, ci53479/co41977#lg41444), -20-p(pear, ci53510/co42000, peach, peach%1:13:00::), -15-p(pear, lg41477, peach, ci53479/co41977#lg41444)]

pear n [] [] ci53510/co42000 [tree, garden, full, apple, trees]
Def: [a sweet juicy fruit with a green skin which has a round base and is slightly pointed towards the stem, a pear is a sweet juicy fruit which is narrow near its stalk and wider and rounded at the bottom pears have white flesh and thin green or yellow skin and grow on trees]

peach n [fruit] [pm, fo] ci53479/co41977#lg41444 [tree, peaches, grown, exported, warm, countries, cream, dessert, tinned, twenty-eight, skin]
Def: [a round fruit with juicy sweet yellow flesh slightly furry red and yellow skin and a large seed in its centre, a peach is a round juicy fruit with sweet yellow flesh and slightly furry yellow and red skin peaches grow in warm countries, a round fruit with soft yellowish-red skin and sweet juicy flesh and having a large rough seed in its centre]

QUERY: nadamas

cputime : 12.7453

Bibliography

"Abroad." Le Grand Robert & Collins Electronique. 2004.

Agirre, E., and P. Edmonds. "Introduction." Word Sense Disambiguation Algorithms and Applications (Text, Speech and Language Technology). Ed. E. Agirre and P. Edmonds. New York: Springer, 2006. 1-28.

Bar-Hillel, Y. "The present status of automatic translation of languages." Advances in Computers. Ed. F. L. Alt. Vol. 1. New York: Academic P, 1960.

Bramer, Max. Logic Programming with Prolog. New York: Springer, 2005.

Cambridge Advanced Learner's Dictionary. Computer software. Vers. 3. 2008.

Concise Oxford English Dictionary. Computer software. Vers. 11. 2004.

Cruse, D. A. Lexical Semantics. Cambridge: CUP, 1986.

Edmonds, P. "Disambiguation, Lexical." Encyclopedia of Language and Linguistics. Vol. III. 2nd ed. 2006. 607-23.

Firth, J. R. "A Synopsis of Linguistic Theory, 1930-55." Selected Papers of J.R. Firth 1952-59. Ed. F. R. Palmer. London: Longmans, 1968. 168-205.

Firth, J. R. "Modes of Meaning." Papers in Linguistics 1934-51. London: Oxford UP, 1957. 190-215.

Gelbukh, Alexander. "Multiword Expressions: A Pain in the Neck for NLP." Computational Linguistics and Intelligent Text Processing Third International Conference, CICLing 2002, Mexico City, Mexico, February 17-23, 2002 Proceedings (Lecture Notes in Computer Science). New York: Springer, 2002. 189-206.

Kilgarriff, A. "I Don’t Believe in Word Senses." Computers and the Humanities 31 (1997): 91-113.

Kilgarriff, A. "Word Senses." Word Sense Disambiguation Algorithms and Applications (Text, Speech and Language Technology). Ed. P. Edmonds and E. Agirre. New York: Springer, 2006. 29-46.

Krishnamurthy, R. "Collocations." Encyclopedia of Language and Linguistics. Vol. II. 2nd ed. 2006. 596-600.

Lea, Diana. Oxford Collocations Dictionary for Students of English. New York: Oxford UP, USA, 2002.

Lesk, M. "Automated sense disambiguation using machine-readable dictionaries: how to tell a pine cone from an ice cream cone." Proceedings of the 1986 SIGDOC conference. Toronto, Canada. 1986. 24-26.

Litkowski, K. C. "Computational Lexicons and Dictionaries." Encyclopedia of Language and Linguistics. Vol. II. 2nd ed. 2006. 753-61.

"Mean." Collins Cobuild. 2002.

Montemagni, S., S. Federici, and V. Pirrelli. "Example-based Word Sense Disambiguation: a Paradigm-driven Approach." Euralex '96 Proceedings. 1996. 151-60.

Oxford Advanced Learner's Dictionary. Computer software. Vers. 7. 2005.

"Roget's Thesaurus." Project Gutenberg. 22 Oct. 2008 <http://www.gutenberg.org/dirs/etext91/roget15a.txt>.

Sanderson, M. "Word Sense Disambiguation and Information Retrieval." Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland. 1994. 142-51.

Voorhees, E. M. "Natural Language Processing and Information Retrieval." Information Extraction: Towards Scalable, Adaptable Systems. Ed. M. T. Pazienza. London: Springer, 1999. 32-48.

Wilks, Y. "Computational Linguistics: History." Encyclopedia of Language and Linguistics. Vol. II. 2nd ed ed. 2006. 761-69.

"WordNet." Princeton University. 26 Oct. 2008 <http://wordnet.princeton.edu/>.