SOLVING VOCABULARY MATCHING PROBLEMS WITH WORDNET

By

QUINTEN YEARSLEY

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree

With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

MAY 2013

Approved by:

Sandiway Fong

[illegible]

Department of Computer Science

The University of Arizona Electronic Theses and Dissertations Reproduction and Distribution Rights Form

Name (Last, First, Middle)
Yearsley, Quinten

Degree title (eg BA, BS, BSE, BSB, BFA):
BS

Honors area (eg Molecular and Cellular Biology, English, Studio Art):
Computer Science

Date thesis submitted to Honors College:
April 29, 2013

Title of Honors thesis:
Solving Vocabulary Matching Problems with WordNet

The University of Arizona Library Release Agreement

[✓] Yes, make my thesis available in the UA Campus Repository!

Student signature: [illegible] Date: 4/29/13

Thesis advisor signature: [illegible] Date: 4/29/13

[ ] No, do not release my thesis to the UA Campus Repository.

Student signature: ______ Date: ______

Last updated: 04/01/13

Abstract

This study investigated the use of the lexical database WordNet to solve vocabulary matching quizzes. By using the relations in WordNet to match words with semantically similar definitions, it is possible to discover current deficiencies in WordNet, and by experimenting with different ways of using WordNet to find matches, some insight was gained into the semantic relations that tend to exist between words and definitions.

Several different methods for measuring semantic similarity between words and definitions were tried and compared, including methods using the WordNet hypernym-hyponym hierarchy, methods using glosses, and methods using other relations in WordNet. Two different algorithms for matching given a set of similarity scores were explored: a greedy matching algorithm and a method that searches globally for the match with the maximum score.

It was discovered that by scoring matches using paths through different WordNet relations, 82-85% of the words in the quizzes could be correctly matched, and the most useful relations were the *hypernym* and *similar to* relations. However, there were many word and definition pairs that were unable to be matched, revealing some places where WordNet could be improved.

Contents

1 Introduction

1.1 Problem Overview

The problem examined in this study is the problem of solving vocabulary quizzes using the lexical database WordNet. In this kind of quiz, one is given a set of words on one side and a set of definitions on the other. The task is to match each word with its corresponding definition. In order to do this, one must assess how likely each word is to match each definition: the more semantically similar a definition is to a word, the more likely it is to be the correct match. An example of such a quiz is shown below:

1	furtive	a	avert an action
2	fortuitous	b	renounce
3	forestall	c	happening by chance
4	gainsay	d	noisy quarrel
5	forswear	e	unruly
6	gambol	f	unwilling to yield
7	froward	g	attack loudly
8	fulminate	h	stealthy
9	fractious	i	contradict
10	fracas	j	frolic

Table 1: Sample vocabulary matching quiz (Quiz 25)

Each word and matching definition phrase are supposed to be synonymous, because the purpose of definitions is to express the same meaning in different words. Therefore, by matching each word with the word or phrase most similar semantically, one should be able to correctly match all of the words and definitions.

This study uses the WordNet, which is a network of data about words and their semantic relations. The relations in WordNet can be used in a variety of ways to assess whether a word should be matched with a particular definition. This study examines the different ways in which the data in WordNet can be used to solve this problem.

1.2 Introduction to WordNet

According to the introduction on the official WordNet website:

> 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. [3]

In WordNet, each sense of a word in WordNet is considered separately, and each sense belongs to a different synonym set, or *synset*, along with senses of other words that are synonymous. If two words share a common sense that has no semantic difference then they belong to the same synset. Each lexical form that is a member of a synset is called a *lemma*.

Synset and lemma relations

A ***hypernym*** of a word W is a word that is more general or broad in meaning than W, and includes W. The converse of a hypernym is a ***hyponym***. For example, *light* is a hypernym of *moonlight*, and conversely *moonlight* is a hyponym of *light*. The hyponym and hypernym relations in WordNet organize nouns and verbs into a kind of taxonomy tree[1].

The two main relationships that connect adjectives in WordNet are ***similar to*** and ***see also***. They both connect adjectives of similar meanings, although the *similar to* relation is more common. The *similar to* relation is reflexive and connects so-called *head adjectives* that have more general meanings with *satellite adjectives* that have more specific meanings. For example, it connects the general adjective *hungry* with the more specific adjectives *ravenous* and *peckish*. The *see also* relation, on the other hand, is not necessarily reflexive, and it connects pairs of adjectives with related meanings, for example *defiant* and *unwilling*.

The ***derivationally-related*** relation links lemmas with different parts of speech that are related by addition or removal of a derivational affix. For example, *insight*

[1]Strictly speaking, verb hyponyms are called troponyms. In this paper, the term hyponym will be used for verbs as well as nouns

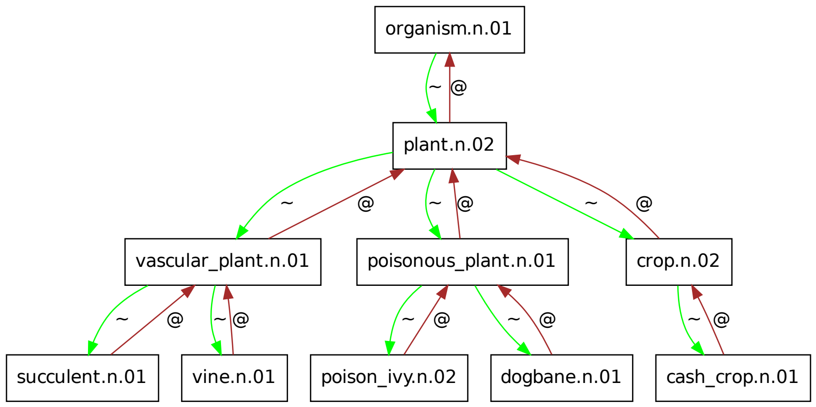

Figure 1: Hypernym-hyponym tree structure
Key: brown (@): hypernym, green (˜): hyponym.

(noun) is derivationally related to *insightful* (adjective), which in turn is derivationally related to *insightfulness* (noun). This is the main relation in WordNet that links words from different lexical categories. It is a reflexive relation.

The ***antonym*** relation connects pairs of lemmas with opposite meanings. Similarly to the *derivationally related* relation, many antonym pairs differ by the addition or removal of an affix. For example, the adjective lemmas *impartial* and *partial* are antonyms, and the verb lemmas *connect* and *disconnect* are antonyms. The antonym relation is reflexive.

There are several other relations, but they are less common in WordNet than those listed above. For descriptions of all of the rest of the relations in WordNet, see `http://wordnet.princeton.edu/`.

1.3 The sample quiz data

In this study, the vocabulary quizzes used were a set of quizzes from *Word Smart for the GRE*, a book designed to help students study words that may appear on the standardized GRE (Graduate Record Examinations) test[1]. They are relatively infrequent words that an average English-speaking adult may not necessarily know. About 50% of the words are adjectives, 30% are nouns, and 20% are verbs. The

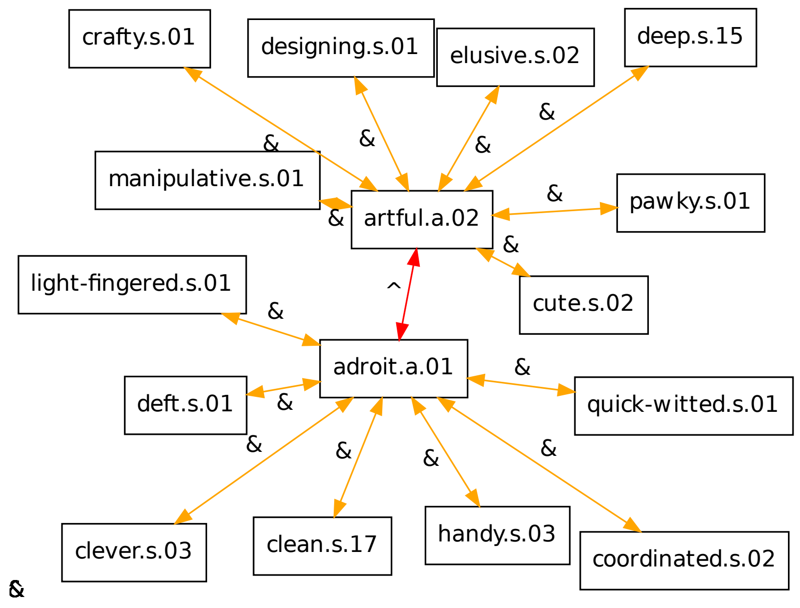

Figure 2: A group of adjectives in WordNet
Key: orange (&): similar to, red (ˆ): see also

definitions use more common words, and consist of either single words or short phrases that are 2-7 words in length.

1.4 Statement of relevance

The first goal of this study is to investigate what types of semantic relationships are useful in matching words and definitions, and to look at how the connections in WordNet can be used for this task. Are definition words usually be found in the same synset as the words they define? Are they more commonly hypernyms or hyponyms, and how useful are the other relations?

The second goal of the study is to help find deficiencies in WordNet. Words and their definitions should generally have close semantic links, then cases where such links cannot be found may indicate links that are missing from WordNet. Automatically searching for cases where links are expected but missing may be used to aid the editors and maintainers of WordNet and improve WordNet.

1.5 Previous Work

Previous researchers have explored various techniques to measure the semantic similarity of word senses using WordNet. Many techniques use the hierarchical structure provided by the hypernym and hyponym relations in WordNet, and apply only to nouns and verbs.

For example, the Wu-Palmer similarity algorithm scores the similarity of two word senses by measuring the depth of the two senses in the taxonomy, as well as the depth of their deepest common ancestor node[5]. The Resnik similarity algorithm measures word sense similarity by looking at the *information content* of the deepest common ancestor node[2]. The application of these and related algorithms to the word-definition matching problem is examined in section 3.2.

2 Methods

2.1 Overall approach

This problem can be broken down into two main subproblems. Firstly, given a word and a definition, how can one score them and assess how likely they are to match? Secondly, once one has a set of scores between the words and definitions, how can one use these scores to find an optimal match between those words and definitions? These two parts are covered in sections 3 and 4.

2.2 Analyzing definition phrases

About 44% of the definitions in the sample quiz data are multi-word phrases, composed of 2-7 words. The phrases have various internal structures, and like the words in the quiz, they can be adjective phrases, verb phrases, or noun phrases. Below are some examples of phrasal definitions in the sample quiz data:

grandiloquence:	pompous speech
torque:	force causing rotation
admonish:	warn or express disapproval

WordNet contains single words and some fixed collocations such as *build up* and *beat around the bush*; it doesn't contain phrases with compositional meanings. So, in order to connect a word with a phrase using WordNet, one must connect it with some individual word or words in the phrase.

One possible approach would be to try to find the head word of the phrase, and then match with that. This could be done by parsing all of the phrases using some external tool, or by guessing based on parts of speech. This would give good results for phrasal definitions where the head word is closely related to the word being matched, for example in the following examples.

censure: criticize severely
laud: praise highly
oscillation: swinging back and forth

However, there are some cases where the word in a phrase that is most likely to match is not the head word of the phrase. In other cases, there are multiple heads, or multiple words in the definition phrase that may be useful in matching. Consider the following definitions:

abate: lessen in intensity
approbation: expression of praise
complaisance: willingness to comply
canon: set of principles
acumen: keen insight

Since a word may often be related to several different words in a definition phrase, good results might be obtained by treating the definition phrase as a collection of words, and trying to match with all of the words in the definition. Of course, function words like *of*, *to*, and *in* aren't likely to have useful relations with the word, so function words were filtered out.

2.3 Tools and libraries

There are WordNet libraries available for a wide variety of programming languages, and they all provide similar functionality. The library I chose for this project was the `nltk.corpus.wordnet` module in the Natural Language Toolkit (NLTK).

Apart from providing an interface to WordNet synsets and all of their relations, NLTK also provides a variety of functions for calculating the similarity of two synsets using methods such as the Wu-Palmer, Resnik, and Leacock-Chodorow methods. The use of these methods for scoring is compared in section 3.2

3 Scoring methods

First we shall look at the different methods used to score matches between words and definitions. In order to compare them, the output of running `match.py` with the greedy matching algorithm and no confidence weighting was compared. For more information about the matching algorithm, see section 4. The source code for each of the methods described in this section can be found in appendix A.2.

3.1 Scoring with synset glosses

Each synset in WordNet has a short gloss to explain the specific meaning of the synset. This gloss does not represent a semantic relation between words; its purpose in WordNet is primarily to explain and differentiate different senses. Since this gloss is a kind of definition, it may contain some words in common with the quiz definition that it should match. So, it could be used to score matches between words and definitions.

The technique used to assess the matches between words and definitions using glosses was to look at the words in the glosses for all of the synsets of the word; the proportion of content words in the definition that were found in any of these glosses was taken as the score for the match. For example, the gloss for the first synset of *laud* is 'praise, glorify, or honor', and the definition in the quiz is 'praise highly'. Since half of the in this definition phrase appear in a gloss, the score is 0.5.

Method	Correct	Total	% Correct
Matched using gloss	197	214	92.06%
Guessing	51	415	12.29%
Total	248	629	39.43%

Table 2: Results when matching using the synset glosses.

These results show that this method did not apply to most of the words; that is, the glosses of the synsets for most words no overlap with any definition phrases. However, when there was some overlap, this was a good indication of a correct match.

3.2 Scoring with similarity functions

Another type of method to score matches between words and definitions using WordNet is by using the distance in the hypernym-hyponym hierarchy. There are several related algorithms that assess the semantic similarity of words, which were briefly mentioned in section 1.5. A summary of the main differences according to the NLTK documentation at `http://nltk.googlecode.com/svn/trunk/doc/api/nltk.corpus.reader.wordnet-module.html` between these methods is listed below.

- **Path distance**: Score based on the shortest path that connects the senses in the hypernym-hyponym taxonomy.
- **Leacock-Chodorow**: Score based on the shortest path that connects the senses and the maximum depth in the taxonomy in which the senses occur.
- **Wu-Palmer**: Score based on the depth of the two senses in the taxonomy and the depth of their least common subsumer (deepest common ancestor node).
- **Resnik**: Score based on the information content of the least common subsumer.
- **Jiang-Conrath**: Score based on the information content of the two input synsets and of the least common subsumer.
- **Lin**: Score based on the information content of the two input synsets and of the least common subsumer.

All of these methods compute the similarity between two synsets, but words in WordNet have a different synset for each different sense. So, in order to score the similarity between a two words using these methods, the maximum score between any pair of synsets of each of the two words was used. In the case of multi-word definitions, the maximum score between the word and any word in the definition was used, as discussed in section 2.2.

For the information content data required by the Resnik, Jiang-Conrath and Lin algorithms, I used the information content data from the NLTK wordnet module. Specifically, the file used was the `ic-brown.dat` file, which was used in the demo function of the `wordnet` module.

Method	# Matched correct	% Matched correct
Path distance	305	48.49%
Leacock-Chodorow	308	48.97%
Wu-Palmer	306	48.65%
Resnik	288	45.79%
Jiang-Conrath	283	44.99%
Lin	287	45.63%

Table 3: Overall scores for each of the similarity functions

Table 3 shows the results of running the matching script on each of these methods. It shows that by using these similarity algorithms, some words can be guessed correctly, but the total score is below 50%. The fact that three methods that used information content scored lower might be due to the small size of the data file used to get information content. These three methods might do better if better data was provided.

3.3 Scoring with synset relation paths

The main problem with the similarity algorithms discussed above is that they don't give any similarity score for adjectives, because adjectives aren't organized into a hypernym-hyponym hierarchy. Additionally, there are other useful relations in WordNet that aren't utilized.

So, the next approach used in scoring the similarity of two synsets was to use the length of the shortest path through WordNet between the two synsets using many different kinds of relations. The first set of relations tried was *hypernym*, *hyponym*, *similar to*, *see also*, *derivationally related*, and *antonym*, because these types of relations are commonly used in WordNet.

As mentioned in section 1.2, some of the relations are between lemmas instead of synsets; for the sake of simplicity, in this implementation a relation between lemmas in two different synsets is considered to be the same as a relation between those two synsets.

The relations in WordNet are between synsets, but in the quiz we are only given lexical forms without sense information. So when searching for paths between two words, this implementation searches all paths between synsets for all senses of both the words. As discussed in section 2.2, a link to any word in a multi-word definition is considered a link to that definition. Additionally, for the sake of simplicity, the score is based only on the shortest path between two synsets.

Scoring scale

For the synset relation path scoring, a scale of 0 to 1 was chosen, where 0 represents no similarity, and 1 represents maximum similarity. This is consistent with the score values given in the other types of scoring methods discussed above, and it also allows the confidences values to be conveniently calculated from the scores, which was discussed in section 3.4.

In order to obtain a value between 0 and 1 given the length L of a path between two synsets, a score S is calculated as $S = 1/(L + 1)$. So, for example, if the two synsets are the same synset, $L = 0$, so $S = 1/(0 + 1) = 1$. If $L = 1$, then $S = 1/(1 + 1) = 0.5$, and so on.

Testing different depths

The paths are found by searching WordNet in a breadth-first manner. In this implementation, a search tree is first built up to a certain depth in a breadth-first manner, and then this search tree is repeatedly searched in a depth-first manner. The depth of the search tree corresponds to the maximum length of the paths that can be found. With very low depths, few matches are found and the score is low. Table 4 shows the results of running the matching script with different depth parameters.

Depth	# Matched correct	% Matched correct	Time taken
1	347	55.17%	0m10.717s
2	436	69.32%	0m20.317s
3	488	77.58%	0m53.447s
4	517	82.19%	2m45.330s
5	527	83.78%	8m53.321s
6	531	84.42%	25m2.202s
7	531	84.42%	57m1.138s
8	531	84.42%	100m3.027s

Table 4: Results when running

Testing different relations

For convenience, each of the different types of relations in WordNet is denoted using a different symbol. These are the symbols used in the WordNet Lexicographer files. Symbols for some of the common relations are listed in table 5. For a full list and more details, see `http://wordnet.princeton.edu/man/wninput.5WN.html`.

Symbol	Relation name
@	hypernym
˜	hyponym
&	similar to
ˆ	see also
+	derivationally related
!	antonym

Table 5: Symbol of common relations used in WordNet Lexicographer files

From table 6, we can see that using just the six relations *hypernym* (@), *hyponym* (˜), *similar to* (&), *see also* (ˆ), *derivationally related* (+), and *antonym* (!), the result is better than if many more relations are used. Using some of the relations results in a lower score, because it creates a shorter path to an incorrect match. For example, *putrefy* has a an entailment relation with *smell*, causing it to match with *having a strong smell* instead of its correct match, *rot*.

Section 5.1 has more discussion about which relations are useful in matching.

free.a.01 [free]

Set of relations	# Matched correct	% Matched correct
{@, ~}	337	53.58%
{@, ~, &, ^}	440	69.95%
{@, ~, &, ^, +, ! }	517	82.19%
{@, ~, &, ^, +, !, > }	518	82.35%
{@, ~, &, ^, +, !, >, \, $, *, =}	508	80.76%
Al	509	80.92%

Table 6: Results of matching with different synset relations (depth 4)

Weighting relations

In the above section, we are measuring semantic distance by counting the number of links in a path through WordNet from one synset to another. However, different relations in WordNet may involve different semantic distances; for example, similar-to links might indicate a closer relation than see-also links. Additionally, different types of links may be statistically less likely to lead from a word to its definition – for example, hyponym links and derivationally-related links may be less likely to lead to a definition word than hypernym links.

Therefore, when calculating the score or cost of a path through WordNet relations, we might get better results if we consider different types of relations as having different costs. However it's hard to guess by just looking at the data what the optimal weights might be.

In order to determine the optimal weights, a list of inequalities was produced, where each inequality reflects an assertion that a correct path should have a lower cost than incorrect paths. For example, in sample quiz 1, the path from *fractious* to the correct match involves 2 *derivationally related* links, whereas a path to an incorrect definition involves 2 *see also* links and 1 *similar to* link. The inequality to reflect this is $2 * D < 2 * A + S$, where D, A and S are variables representing costs of different types of relations.

After generating all such inequalities, a Prolog program was used to search for assignments of the variables that result in the largest set of consistent inequalities. One set of weights was found that resulted in all the inequalities being consistent for a certain subset of the quizzes. The set of sample quizzes was

$[3, 9, 11, 12, 13, 14, 15, 26, 29, 30, 31, 32, 33, 35, 40, 42, 46, 47, 51, 58, 64]$, and the set of weights is listed in table 7.

Relation	Weight
Hypernym	3
Hyponym	2
Similar to	1
See also	4
Derivationally related	4
Antonym	5

Table 7: Weights resulting in consistent inequalities for a subset of the quizzes

Weighting	# Matched correct	% Matched correct
No	160	83.33%
Yes	156	81.25%

Table 8: Comparison of results for quiz subset with and without weights

The results of running this set of quizzes with and without weighting of the different relations is shown in table 8. Surprisingly, using the weighting resulted in a slightly lower score. This may be because the actual ranking and matching does not directly use the length of the path L, but instead uses the value $S = 1/(L + 1)$.

3.4 Confidence weighting

Some words match with only one definition, but others match with several definitions. In such a case, one can be more confident about matching that word that matches only one definition than when matching the word that matches several definitions. For example, in sample quiz 4, the word *apposite* only has one path to a definition word, whereas *apprise* has paths to several words in different definitions, as shown in figure 3. Because there are several possible matches, the certainty about *apprise* is slightly lower. On the other hand, the similarity score between the word and definition is still the most important fact in determining how certain a match is.

To take this factor into account, the matches can be assigned a weighted confidence score after all of the matches for a word are found. Given S, the similarity

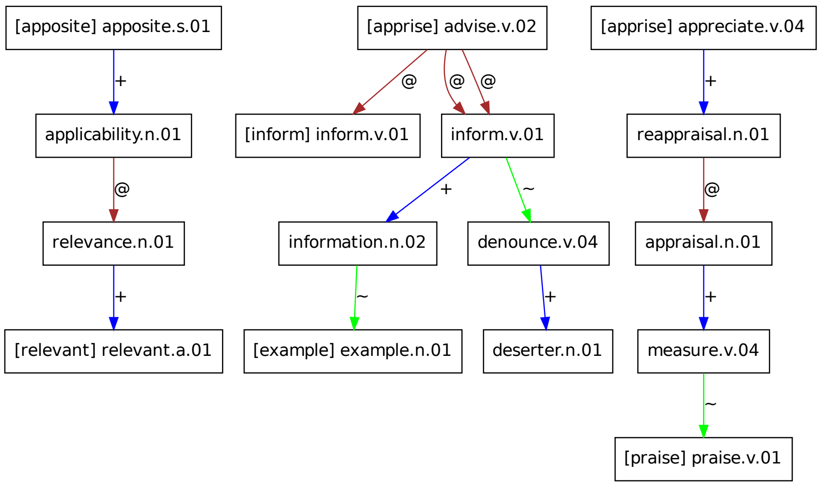

Figure 3: Some links between words in quiz 4
Key: brown (@): hypernym, green (˜): hyponym, blue (+): derivationally-related

score for an individual match, and T, the total of all scores for all matches involving that word, a confidence value C is computed as $C = (S/T)S = S^2/T$. The value S/T is the proportion of one score among all the scores; for example, if there are two equally good matches, this is 0.5. If there is only one match, this is 1.0. However, the score value is still the most important factor in determining how confident one is in a match, so this ratio is then multiplied by score value to give the confidence value.

Confidence value C =	# Matched correct	% Matched correct
S	517	82.19%
S/T	508	80.76%
S^2/T	521	82.83%
S^3/T	522	82.99%
S^4/T	518	82.35%

Table 9: Comparison of results with and without confidence weighting

Once this confidence value is obtained for each match, this confidence value can be used as the score value for each match. Table 9 shows the results of running the matching script with the synset relation paths of up to length 4, with the greedy matching algorithm described in section 4.1. It shows that slightly better results were obtained by using confidence values that are waited in the method above, although the amount of improvement wasn't extremely great.

```
Default (C = S)                       With confidence weighting (C = S^2/T)
Quiz 22:                              Quiz 22:
1. expiate: atone for [1.00]          1. exemplar: model [1.00]
2. exemplar: model [1.00]             2. expiate: atone for [1.00]
3. *extirpate: remove obscenity [0.50] 3. exigent: urgent [0.33]
4. *exhort: urgent [0.50]             4. exonerate: remove blame [0.25]
5. exonerate: remove blame [0.25]     5. *extirpate: remove obscenity [0.50]
6. expatiate: discuss at length [0.25] 6. expatiate: discuss at length [0.25]
7. *extemporaneous: incite [0.00]     7. exhort: incite [0.33]
8. extant: not destroyed [0.00]       8. *extemporaneous: not destroyed [0.00]
9. *exigent: destroy [0.00]           9. *extant: destroy [0.00]
10. *expurgate: improvised [0.00]     10. *expurgate: improvised [0.00]
```

Figure 4: Comparison of the matching results of sample quiz 22

Figure 4 shows the output for a sample quiz, where the order of word matching was different depending on whether confidence weighting was used. In the output on th left, words with higher absolute scores are always matched before those with lower scores. In the output on the right, words are matched in order of confidence rather than absolute score. In this particular quiz, the word *exhort* had a close match with *urgent*, so these two words were matched, blocking the correct word *exigent* from matching with *urgent*. In the output on the right, the match of word *exigent* and *urgent* has a higher confidence, so they're matched first, resulting in *exhort* also being correctly matched with *incite*.

4 Matching methods

Once one has a set of similarity scores between words and definitions, it's not necessariliy straightforward to decide how best to match them. In this section, two methods for matching are explored. Source code for the main matching program, including the algorithms discussed in this section, is available in appendix A.1.

4.1 Greedy matching

The first matching algorithm that was tried is based on the way that people sometimes solve matching problems: One may first find the word and definition that one is most confident about, and draw a line between them. Once that word and definition have already been matched, they can be disregarded, and one can move on to matching the words and definitions that one is less confident about. By the time one matches the item that is most uncertain about, all the other more certain choices have already been eliminated.

This algorithm starts with a list of word and definition candidates, and a list of score values between words and definitions, and proceeds as follows:

1. Find the best score in the set of scores, and record a match between that word and that definition.
2. Remove that word and definition from the lists of candidates, and remove all scores that involve that word or definition.
3. If there are still scores remaining, go back to step 1.
4. Finally, the remaining words and definitions are matched randomly.

The result is that the words with the highest scores are matched first. The results given by this algorithm are usually fairly good, but not necessarily optimal. If a word has a high score for a match with an incorrect definition, and it gets matched with that definition, then that definition is taken out of consideration for the rest of the matching, which may cause other words further down the list to be mismatched.

4.2 Global optimal matching

The goal when matching is to find the correspondence that has the maximum likelihood of being correct. The higher the sum of all of the score values of all corresponding words and definitions, the more confident one is in the correspondence. The global optimal matching algorithm searches for the correspondence with the highest sum of score values. This can be done using brute force, by trying to match every possible permutation of definitions with the same sequence of words. By exhaustively searching through the space of all possible matchings, the matching that gives the maximum score value sum can be found.

However, this is extremely slow. On average, there are about 10 words per quiz, and some quizzes have as many as 13 or 14 words. The number of permutations to be tried in a quiz with n words is $nfactorial$. So, for a quiz with 14 words, there are $14! = 87178291200$ permutations to try.

The global matching algorithm would not complete after several hours of running, I tried testing it on only sample quizzes 1-15, which don't contain any quizzes with length greater than 10. I timed the runtime with the Unix command `time`. Table 10 shows that on sample quizzes 1-15, the global matching algorithm matched a total of 3 more words correctly than the greedy algorithm, but it took much longer to run.

Matching strategy	# Matched correct	% Matched correct	Time taken
Greedy	110	80.88%	0m54.820s
Global	113	83.09%	36m49.998s

Table 10: Comparison of the greedy and global matching algorithms

Figure 5 is an example of a case where the global matching algorithm does better. Using greedy matching, the word *apotheosis* was matched with *expression of praise*, preventing the correct word *approbation* from being matched with *expression of praise*. However, using global matching, the correspondence with the highest total score was chosen, resulting in these two words being correctly matched.

```
Greedy (total score: 2.28)                Global (total score: 2.483)
Quiz 4:                                   Quiz 4:
1. apostate: one who abandons... [0.50]   1. appropriate: confiscate [0.25]
2. apprise: inform [0.50]                 2. apostate: one who abandons faith [0.50]
3. *apotheosis: expression of... [0.33]   3. apotheosis: perfect example [0.20]
4. appropriate: confiscate [0.25]         4. apogee: zenith [0.25]
5. apogee: zenith [0.25]                  6. approbation: expression of praise [0.33]
6. apposite: relevant [0.25]              5. *antithetical: spurious [0.00]
7. antithetical: diametricall... [0.20]   7. apposite: relevant [0.25]
8. *approbation: spurious [0.00]          8. apprise: inform [0.50]
9. *apocryphal: perfect example [0.00]    9. *apocryphal: diametrically opposed [0.20]
```

Figure 5: Comparison of the matching results of sample quiz 4

However, the global optimal matching algorithm doesn't necessarily perform better than the greedy matching algorithm in all cases. Figure 6 shows an example of a case where it doesn't do as well. In the greedy algorithm, *aver* is matched with *state as fact* because it has a short path to *state*. However, *aver* also has a match with another word, and *axiom* also has a match with *state*, resulting in the match with the highest total score actually having more incorrect matches. Figure 7 shows some of the paths through WordNet that result in this matching.

```
Greedy (Total score: 4.2)                 Global (Total score: 4.366)
Quiz 7:                                   Quiz 7:
1. axiomatic: self-evident [1.00          1. axiomatic: self-evident [1.00]
2. baleful: sinister [1.00]               2. baleful: sinister [1.00]
3. avarice: greed [1.00]                  3. *aver: source of harm [0.25]
4. aver: state as fact [0.33]             4. auspicious: favorable [0.33]
5. auspicious: favorable [0.33]           5. *beatify: universally recognized principle [0.20]
6. austere: bare [0.33]                   6. *axiom: state as fact [0.25]
7. *beatify: universally recog... [0.20]  7. austere: bare [0.33]
8. *axiom: source of harm [0.00]          8. avarice: greed [1.00]
9. *bane: regard of saintly [0.00]        9. *bane: regard of saintly [0.00]
```

Figure 6: Comparison of the matching results of sample quiz 7

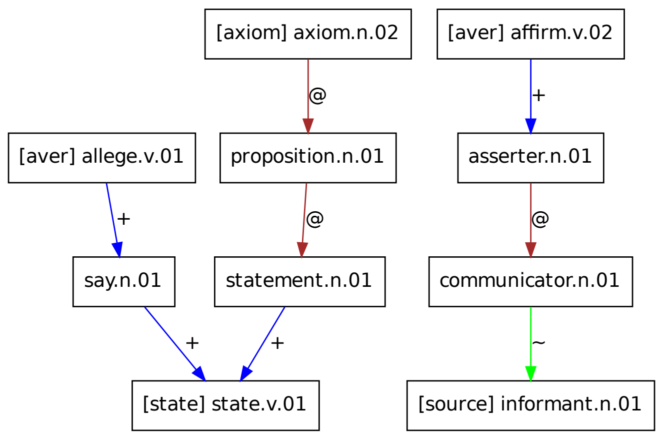

Figure 7: Links between a few key words in quiz 7
Key: brown (@): hypernym, green (˜): hyponym, blue (+): derivationally-related

5 Discussion

5.1 Semantic relationships between words and definitions

What types of connections in WordNet are successful in matching words with definitions? The following tables are taken from the output of running `match.py` with all the default parameters, although the results are similar when run with different parameters.

Relation	Correct	Incorrect	Total	% Correct
1 hypernym	69	4	73	94.52%
1 similar to	57	1	58	98.28%
1 deriv.	17	6	23	73.91%
1 see also	7	0	7	100.00%
1 hyponym	4	0	4	100.00%

Table 11: Matches using single-relation paths

The single one-link relation that was used to match the most words was the hypernym relation, producing 69 correct matches. Compare this with its converse, the hyponym relation, which only produced 4 correct matches. This may be because definitions often use more general words, or because many of the phrasal definitions consist of a modifier and a more general head word.

As shown in table 13, 133 out of 629 words were matched with a definition word that was in a common synset. This indicates that although many of the definition words in the quizzes were synonymous with the word that they matched, most of the words were not actually perfect synonyms; oftentimes the definitions in the quizzes were just supposed to be closely related in meaning, but not necessarily synonymous.

It's interesting to note that one of the words matched with a definition in the same synset was actually an incorrect match. In this case, the word *denigrate* was supposed to match with *disparage*, and *deprecate* was supposed to match with *belittle*. This was particularly tricky, because the words *denigrate*, *disparage*, *belittle*, and *disparage* are all closely related semantically and closely linked in WordNet.

Relations	Correct	Incorrect	Total	% Correct
2 deriv., deriv.	24	1	25	96.00%
2 similar to, similar to	20	0	20	100.00%
2 hypernym, hypernym	15	0	15	100.00%
2 hypernym, hyponym	14	4	18	77.78%
2 hypernym, deriv.	7	0	7	100.00%
2 deriv., hypernym	5	1	6	83.33%
2 similar to, see also	5	0	5	100.00%
2 see also, similar to	4	0	4	100.00%
2 similar to, antonym	2	0	2	100.00%
2 similar to, deriv.	1	0	1	100.00%
2 deriv., hyponym	1	0	1	100.00%
2 antonym, similar to	1	0	1	100.00%
2 see also, see also	1	0	1	100.00%
2 see also, hypernym	0	1	1	0.00%

Table 12: Matches using double-relation paths

Relation	Correct	Incorrect	Total	% Correct
Same synset	133	1	134	99.25%
None	32	66	98	32.65%
Length 4 paths	42	17	59	71.19%
Length 3 paths	55	9	64	85.94%

Table 13: Matches not involving a path of length 1 or 2

5.2 Deficiencies in WordNet

Because the words and definitions are supposed to be roughly synonymous, testing whether WordNet has any semantic connections between each word and definition is one way to find possible gaps in the current relations in WordNet.

The scoring functions in `score.py` were used to try to match the words and definitions that are known beforehand to be correct matches. For some multi-word definitions, it's understandable or expected that there shouldn't be a match, because no particular one word in the definition has a close relationship with the defined word, for example in the following definitions:

recondite: difficult to understand

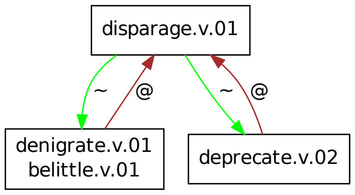

Figure 8: A group of similar verbs
Key: orange (&): similar to, red (ˆ): see also

mitigate:	make less severe
reconnoiter:	make preliminary inspection

However, for many of the words, it's clear that there is a close semantic relationship between the word and definition, and this fact should be reflected in WordNet. Below is a list of some suggested changes to WordNet

Nouns

Failed match:	**dynamo: energetic person**
Suggested change:	Add a sense to the word dynamo to include person sense of the word.
Dictionary entry:	”dynamo (2): a forceful energetic individual”[4]

Failed match:	**desuetude: disuse**
Suggested change:	Add a hypernym relation from disuse#1 to inaction#1, making disuse#1 and desuetude#1 sister synsets.
Relevant synsets:	(n) desuetude#1 (a state of inactivity or disuse) (n) inaction#1 (the state of being inactive) (n) disuse#1 (the state of something that has been unused and neglected)

Failed match:	**equanimity: self-possession**
Suggested change:	Add a hypernym relation between equanimity#1 and self-possession#1.
Relevant synsets:	(n) equanimity#1 – (steadiness of mind under stress) (n) self-possession#1 (the trait of resolutely controlling your own behavior)

Failed match:	**guy: rope used to guide**
Suggested change:	Add a hypernym relation from guy#3 to rope#1
Relevant synsets:	(n) guy#3 (a cable, wire, or rope that is used to brace something) (n)rope#1 (a strong line)

Failed match:	**sanctimony: self-righteousness**
Suggested change:	Add a sense for self-righteousness, which should be a member of the sanctimony#1 synset.
Dictionary entry:	”sanctimony(2): affected or hypocritical holiness”[4]
Relevant synsets:	(n) sanctimony#1 (the quality of being hypocritically devout)

Verbs

Failed match:	**adumbrate: foreshadow**
Suggested change:	Add a sense to adumbrate, which should be a troponym of foreshadow#1.
Dictionary entry:	”adumbrate (1): to foreshadow vaguely”[4]
Relevant synsets:	(v) foreshadow#1 (indicate, as with a sign or an omen)

Failed match:	**broach: bring up**
Suggested change:	Add hypernym relation from broach#1 to bring_up#6, or merge the two synsets.
Relevant synsets:	(v) broach#1 (bring up a topic for discussion)
	(v) bring up#6 (put forward for consideration or discussion)

Failed match:	**impugn: attack verbally**
Suggested change:	Add a hypernym relation from inpugn#1 to attack#2.
Relevant synsets:	(v) impugn#1 (attack as false or wrong)
	(v) attack#2 (attack in speech or writing)

Failed match:	**regale: entertain**
Suggested change:	Add a hypernym relation from regale#1 to entertain, or add a sense to regale#1 that has a hypernym relation to entertain#1.
Dictionary entry:	”regale 1: to entertain sumptuously”[4]
Relevant synsets:	(v) regale#1 (provide with choice or abundant food or drink)
	(v) entertain#1 (provide entertainment for)

Failed match:	**repine: long for**
Suggested change:	Add a sense for repine, and add a sense for long_for, which is currently not in WordNet. The new sense of repine and long_for should be in the same synset.
Dictionary entry:	”repine: to long for something”[4]

Adjectives

Failed match:	**hirsute: shaggy**
Suggested change:	Add a sense to shaggy which is a member of the synset hirsute#1 or has a similar-to or see-also relation with it.
Dictionary entry:	”shaggy (1a): covered with or consisting of long, coarse, or matted hair”[4]
Relevant synsets:	(adj) hirsute#1 (having or covered with hair) (adj) shaggy#1 (used of hair; thick and poorly groomed)

Failed match:	**inchoate: unformed**
Suggested change:	Add a similar-to or see-also relation from inchoate#1 to unformed#2.
Relevant synsets:	inchoate#1 (only partly in existence; imperfectly formed) (adj) unformed#2 (not formed or organized)

Failed match:	**mercurial: unpredictable**
Suggested change:	Add a similar-to or see-also relation between mercurial#1 and unpredictable#1
Relevant synsets:	(adj) mercurial#1 (liable to sudden unpredictable change) (adj) unpredictable#1 (not capable of being foretold)

Failed match:	**morose: melancholy**
Suggested change:	Add a similar-to relation between morose#1 and melancholy#1.
Relevant synsets:	(adj) morose#1 (showing a brooding ill humor) (adj) melancholy#1 (characterized by or causing or expressing sadness)

Failed match:	**variegated: multicolored**
Suggested change:	Add a similar-to relation between variegated#1 and multicolored#2, or add a sense to multicolored which is a member of variegated#1.
Relevant synsets:	(adj) variegated#1 (having a variety of colors) (adj) multicolored#1 (having sections or patches colored differently and usually brightly)

The above list shows some suggestions based on a few of the word-definition pairs in the sample quizzes. Some of the other word-definition pairs that didn't have any links but could be expected to have links include:

ennui: restlessness
harrow: distress
qualms: reservations
depredate: plunder
inimical: harmful

irascible: temperamental
mercurial: unpredictable
sodden: soaked
sophomoric: immature
ubiquitous: widespread

5.3 Future work

The last explored just one method of finding holes in WordNet. In general, any list of pairs of words that is expected to be semantically similar could be used to test for holes in in WordNet, using a manner similar to what was used in the above section.

Additionally, information internal to WordNet could also be used to find potential gaps in WordNet. In many of the cases listed above, an important related word was listed in the gloss of a synset, but there were no actual relations in WordNet linking to that word. A program could be written to search automatically for cases such as these, in order to help improve WordNet.

As for the problem of using the existing information in WordNet to solve vocabulary quizzes, there are several things that could be done in future work. The presence of multiple paths between two synsets and the parts of speech of the synsets could be taken into account, and definition phrases could be parsed and analyzed to provide more information; all of these could potentially lead to better results.

References

[1] Anne Curtis. *Word Smart for the GRE*. Princeton Review Publishing, 2003.

[2] P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. *Arxiv preprint cmp-lg/9511007*, 1995.

[3] Princeton University. About wordnet, 2013.

[4] Merriam-Webster.com, 2013.

[5] Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

A Source code

In this appendix, the source code for the main modules of the program, `match.py` and `score.py`, are listed below for reference. Full source code for the whole program, along with sample output, can be found at `http://qeny.net/thesis2013/`.

A.1 match.py

```python
#!/usr/bin/python
"""
Module: match
Author: Quinten Yearsley
Date: 2013-05-01
Usage: match.py [-h] [-q QUIZ_ID] [-d DEPTH] [-m METHOD] [-c] [-v]

This is the main module for running the word-definition matching program.
It reads in the quiz file and attempts to match words and definitions from
the quizzes, keeping track of methods used to score words and definitions.
"""

import argparse
import nltk.data
import nltk.corpus
import readfuncs
import score

def main():
    """The main function for this module. Read in the quiz file and answers
    file, try to match the words and defintions, and output the results.
    Specific behavior is determined by command-line arguments.
    """
    # Get and parse command line arguments.
    argparser = argparse.ArgumentParser(description="Match words in quizzes.")
    argparser.add_argument('-q', '--quiz-id', type=int, default=0)
    argparser.add_argument('-d', '--depth', type=int, default=4)
    argparser.add_argument('-m', '--method', type=str, default="Path")
    argparser.add_argument('-c', '--confweight', action='store_true')
    argparser.add_argument('-v', '--verbose', action='store_true')
    argparser.add_argument('-g', '--globalmatch', action='store_true')
    args = argparser.parse_args()

    # read_quizfile returns a dictionary of quiz ids to tuples of lists of
    # words and definitions (each tuple has the data for one quiz).
    id_to_quiz = readfuncs.read_quizfile("GRE_quizzes.txt")

    # read_answerfile returns a dictionary of quiz ids to dictionaries of
    # words to definitions.
    id_to_answer = readfuncs.read_answerfile("GRE_answers.txt", id_to_quiz)

    quiz_ids = [args.quiz_id] if args.quiz_id else id_to_quiz.keys()
    rel2weights = {}

    id_to_myanswers = {}
    id_to_scores = {}

    for quiz_id in sorted(quiz_ids):
        words, sdefs = id_to_quiz[quiz_id]
        matches = match(words, sdefs, method=args.method, depth=args.depth,
                        weights=rel2weights, confweight=args.confweight,
                        globalmatch=args.globalmatch)
        word_to_myanswer, word_to_score, words = matches
        id_to_myanswers[quiz_id] = word_to_myanswer
        id_to_scores[quiz_id] = word_to_score
        word_to_correct_sdef = id_to_answer[quiz_id]
        print "Quiz {0}:".format(quiz_id)
        for n, word in enumerate(words, start=1):
            sdef = word_to_myanswer[word]
            is_correct = (sdef == word_to_correct_sdef[word])
            score = word_to_score[word]
            star = "" if is_correct else "*"
            if args.verbose:
                formatstr = "{0}. {1}{2}: {3}, {4}"
                print formatstr.format(n, star, word, sdef, str(score))
            else:
                formatstr = "{0}. {1}{2}: {3}"
                print formatstr.format(n, star, word, sdef)
        print
    if args.verbose:
        score_stats(id_to_myanswers, id_to_scores, id_to_answer)


def score_stats(id_to_answer, id_to_scores, id_to_ref_answer):
    """Print out statistics about the score methods that were used, including
    how many times they were used and how often they yielded correct answers.
    """
    # The first section of this function gathers information about how often
    # each method was used and how many times it resulted in a correct match.
    method_correct_count = {} # Number of times resulted in correct match.
    method_count = {} # Number of uses of each method overall.
    for id in id_to_answer:
        word_to_sdef = id_to_answer[id]
        word_to_score = id_to_scores[id]
        word_to_ref_sdef = id_to_ref_answer[id]
        for word in word_to_sdef:
            correct = (word_to_sdef[word] == word_to_ref_sdef[word])
            method = word_to_score[word].method
            if method not in method_count:
                method_count[method] = 0
                method_correct_count[method] = 0
            method_count[method] += 1
            if correct:
                method_correct_count[method] += 1

    # The second part prints out the information gathered above in a table.
    formatstr = "{0:15}␣|␣{1:5}␣|␣{2:5}␣|␣{3:5}␣|␣{4:6}"
    hbar = "-" * 40
    print formatstr.format("Method␣name", "Right", "Wrong", "Total", "%Right")
    print hbar
    formatstr = "{0:15}␣|␣{1:5}␣|␣{2:5}␣|␣{3:5}␣|␣{4:.2%}"
    for method in reversed(sorted(method_count)):
        total = method_count[method]
        right = method_correct_count[method]
        wrong = total - right
        perc = float(right) / total
        print formatstr.format(method, right, wrong, total, perc)
    total_uses = sum(method_count.values())
    total_right = sum(method_correct_count.values())
    total_wrong = total_uses - total_right
    total_perc = float(total_right) / total_uses
    print hbar
    print formatstr.format("Total", total_right, total_wrong, total_uses,
                           total_perc)


def match(words, sdefs, method="Path", depth=4, weights={}, confweight=False,
          globalmatch=False):
    """Find matches for the words and short definitions of one quiz.

    @param words: List of words in this quiz
    @param sdefs: List of short definitions in this quiz
    @param depth: Maximum depth to search for each word

    @return: A 3-tuple that contains a word to definition dict, a word to score
    dict, and a list of a word in the order that they were matched.
    """
    if globalmatch:
        return global_optimal_match(words, sdefs, method=method, depth=depth,
          weights=weights, confweight=confweight)

    confs = wordconfs(words, sdefs, method=method, depth=depth, weights=weights,
                      confweight=confweight)
    confs.sort(reverse=True)

    # The following variables will be used to hold the matching results.
    word_to_sdef, word_to_score = {}, {}
    matched_words, matched_sdefs = [], []

    # Look at the scores in the (conf, score) list and match the words and
    # definitions that have the highest confidence score first.
    for _, s in confs:
        if s.word in matched_words or s.sdef in matched_sdefs:
            continue
        word_to_sdef[s.word] = s.sdef
        word_to_score[s.word] = s
        matched_words.append(s.word)
        matched_sdefs.append(s.sdef)

    # For all remaining words, since they couldn't be matched above,
    # match them together randomly.

    words = filter(lambda w: w not in matched_words, words)
    sdefs = filter(lambda d: d not in matched_sdefs, sdefs)
    for i in range(len(words)):
        word, sdef = words[i], sdefs[i]
        word_to_score[word] = score.Score(word, sdef)
        word_to_sdef[word] = sdef
        matched_words.append(word)

    return (word_to_sdef, word_to_score, matched_words)


def global_optimal_match(words, sdefs, method="Path", depth=4, weights={},
                         confweight=False):
```

```python
    """Same as the above match but matches using the global optimal match
    method instead of the greedy matching algorithm.
    """
    confs = wordconfs(words, sdefs, method=method, depth=depth, weights=weights,
                      confweight=confweight)
    high_score = 0
    best_match = {}
    best_word_to_score = {}
    for i, sdef_permutation in enumerate(xpermutations(sdefs)):
        this_match = dict(zip(words, sdef_permutation))
        this_score, this_word_to_score = total_score(this_match, confs)
        if this_score > high_score:
            high_score = this_score
            best_match = this_match
            best_word_to_score = this_word_to_score
    print high_score
    return (best_match, best_word_to_score, words)


def xpermutations(items):
    """Generate all permutations of items in the given list."""
    return xcombinations(items, len(items))

def xcombinations(items, n):
    """Generate all combinations of length n of items in the given list."""
    if n == 0:
        yield []
    else:
        for i in xrange(len(items)):
            for cc in xcombinations(items[:i]+items[i+1:],n-1):
                yield [items[i]]+cc

def total_score(word_to_sdef, confs):
    """Assess an assignment between a list of words and list of definitions.

    @param word_to_sdef: A dict of word to short definition.
    @param confs: A list of (confidence value, Score) tuples.
    @return: A Tuple of total score value, and dict of word to Score.
    """

    # A dictionary of (word,score) pairs to (confidence value, Score)
        pairs
    confscore_dict = dict([((s.word, s.sdef), (conf, s)) for conf, s
        in confs])

    total = 0
    word_to_score = {}
    for word, sdef in word_to_sdef.items():
        if (word, sdef) in confscore_dict:
            conf, s = confscore_dict[word, sdef]
            total += conf
            word_to_score[word] = s
        else:
            word_to_score[word] = score.Score(word, sdef, value=0)

    return (total, word_to_score)


def wordconfs(words, sdefs, method="Path", depth=4, weights={},
              confweight=True):
    """Make a list of all the scores between all words and definitions
        .
    If the confweight flag is set to true, then the confidence will be
         altered
    based on how many different matches there are for a given word.

    @return: A List of (confidence value, Score) tuples.
    """
    confs = []
    for word in words:
        scores = wordscores(word, sdefs, method=method, depth=depth,
                            weights=weights)
        if confweight:
            wordconfs = weighted_confidences(scores)
        else:
            wordconfs = [(s.value, s) for s in scores]
        confs.extend([(v, s) for (v, s) in wordconfs if v > 0])
    return confs


def wordscores(word, sdefs, method="Path", depth=4, weights={}):
    """Get all the scores for all matching definitions for a given words.

    @param word: The word to match.
    @param sdefs: The list of short definitions to match.
    @return: A list of Score objects for all matches with the given word.
    """
    scores = []
    if method == "Path":
        tree = score.SearchTree(word)
        for depth in xrange(depth):
            tree.deepen()
        for sdef in sdefs:
            scores.append(tree.score(sdef, weights))
    elif method == "Gloss":
        for sdef in sdefs:
            scores.append(score.gloss_score(word, sdef))
    else:
        path_pointer = nltk.data.find("corpora/wordnet_ic")
        wnic = nltk.corpus.WordNetICCorpusReader(path_pointer, ".*\.dat")
        ic = wnic.ic("ic-brown.dat")
        for sdef in sdefs:
            scores.append(score.similarity_score(word, sdef, method, ic))
    return scores


def weighted_confidences(scores):
    """Calculate confidence values for a list of scores based on the value of
    each score as well as the total value of all scores in the list

    @param scores: list of the scores for all the matches with a word
    @return: A list of tuples of confidence value and Score object.
    """
    scores = [s for s in scores if s.value > 0]
    totalvalue = sum([s.value for s in scores])
    if totalvalue == 0:
        return []
    confs = []
    for s in scores:
        confidence = s.value * s.value / totalvalue
        confs.append((confidence, s))
    return confs


# If this module is run directly and not imported, run the main function.
if __name__ == "__main__":
    main()
```

A.2 score.py

```
#!/usr/bin/python
"""
Module: score
Author: Quinten Yearsley
Date: 2013-05-01

This module contains functions relating to scoring the
relatedness of words and definitions.

The main method for scoring (scoring with paths) involves building a search
tree of paths from the given node, and so this module also contains the
classes and methods related to these search trees and paths, as well as the
Score class.
"""

from nltk.corpus import wordnet
import nltk.data
import nltk.corpus

# A list of synset and lemma relations that will be searched
RELATIONS = [
    '@',  # hypernyms
    '~',  # hyponyms
    '&',  # similar_tos
    '^',  # also_sees
    '+',  # derivationally_related_forms
    '!',  # antonyms
]

# A list of function words that will be removed from multi-word
    defintions
STOPLIST = [
    "the", "a", "an", "to", "of", "at", "in", "into", "on", "onto", "
        from",
    "by", "for", "with", "around", "up", "down", "or", "and", "any", "
        all",
    "one", "who", "that", "which", "as", "like", "only", "just", "not"
        ,
    "be", "come", "go", "give", "take", "get", "make"
]

def score(word, sdef, depth=4):
    """Score the similarity of a word and sdef using a SearchTree."""
    tree = SearchTree(word)
    for i in range(depth):
        tree.deepen()
    return tree.score(sdef)

def similarity_score(word, sdef, method, ic):
    """Score a word and sdef using the NLTK similarity functions."""
    word_synsets = wordnet.synsets(word)
    methods = {
        "path_similarity": lambda s1, s2: s1.path_similarity(s2),
        "lch_similarity": lambda s1, s2: s1.lch_similarity(s2),
        "wup_similarity": lambda s1, s2: s1.wup_similarity(s2),
        "jcn_similarity": lambda s1, s2: s1.jcn_similarity(s2, ic),
        "lin_similarity": lambda s1, s2: s1.lin_similarity(s2, ic),
        "res_similarity": lambda s1, s2: s1.res_similarity(s2, ic)
    }

    sdef_synsets = []
    for sdef_word in content_words(sdef):
        sdef_synsets.extend(wordnet.synsets(sdef_word))
    max_score = 0
    details = ""
    for synset1 in word_synsets:
        for synset2 in sdef_synsets:
            if synset1.pos not in ["n", "v"] or synset1.pos != synset2
                .pos:
                continue
            this_score = methods[method](synset1, synset2)
            if this_score > max_score:
                max_score = this_score
                details = "\t" + synset1.name + "␣:␣" + synset2.name
    return Score(word, sdef, value=max_score, method=method, details=
        details)

def gloss_score(word, sdef):
    """Score the similarity of a word and definition by counting the
        proportion
    of words in the definition that show up in the glosses of the word
        's
    different senses.
    """
    glosses = "␣".join([s.definition for s in wordnet.synsets(word)])
    word_set = set(content_words(glosses))
    sdef_set = set(content_words(sdef))
    intersect_size = len(word_set.intersection(sdef_set))
    scorevalue = float(intersect_size) / len(sdef_set)
    details = "Glosses:␣"
    details += ";␣".join(([s.definition for s in wordnet.synsets(word)
        ]))
    return Score(word, sdef, value=scorevalue, method="Gloss", details
        =details)

def content_words(phrase):
    """Get a list of non-function words from a string,
```

```python
    by removing punctuation and words that are in the STOPLIST.
    """
    phrase = phrase.translate(None, ":;,.!()") # Remove punctuation
    if "␣" not in phrase:
        return [phrase]
    words = [w for w in phrase.split() if w not in STOPLIST]
    return words


class SearchTree:
    """A SearchTree contains the information required to search from a word
    to synsets that are related by a series of links. Since each word has
    multiple sense and thus multiple synsets, a SearchTree actually contains
    multiple trees whose roots are the synsets of the starting word.
    """

    def __init__(self, word):
        """Create a search tree from the given word."""
        self.word = word
        start_synsets = wordnet.synsets(word)
        self.children = [Node(s) for s in start_synsets]
        self.leaves = []
        for node in self.children:
            self.leaves.append(node)
        self.synsets = set(start_synsets) # all the synsets in the tree

    def deepen(self):
        """Deepen the tree by one level. """
        new_leaves = []
        for leaf in self.leaves:
            for relation in RELATIONS:
                related = leaf.synset._related(relation)
                lemmas = leaf.synset.lemmas
                related_lemmas = [lemma._related(relation) for lemma in lemmas]
                related_lemmas = sum(related_lemmas, []) # flatten
                related.extend([lemma.synset for lemma in related_lemmas])
                new_related = [r for r in related if r not in self.synsets]
                for related_synset in new_related:
                    child = Node(related_synset)
                    leaf.children.append((relation, child))
                    new_leaves.append(child)
                    self.synsets.add(related_synset)
        self.leaves = new_leaves

    def score(self, sdef, weights={}):
        """Score a word and short definition match based on this tree.
        Return a Score object, which represents the score and related info.
        """
        if " " not in sdef: # Single-word definitions
            sdef_words = [sdef]
        elif wordnet.synsets(sdef.replace(" ", "_")): # Multi-word collocations
            sdef_words = [sdef.replace(" ", "_")]
        else: # Phrasal definitions
            sdef_words = content_words(sdef)
        paths = []
        for w in sdef_words:
            paths += search_paths(self, w)
        return Score(self.word, sdef, paths=paths, weights=weights)

    def __str__(self):
        """Make a string showing the contents of the SearchTree."""
        result = "SearchTree: " + self.word + "\n"
        for child in self.children:
            result += "\t" + str(child) + "\n"
        return result


class Node:
    """A Node object represents a synset and its links to other nodes, which
    represent related synsets. The children list is a list of pairs of
    relation strings and synsets.
    """
    def __init__(self, synset):
        """Initialize a new Node from the given synset with no links.
            """
        self.synset = synset
        self.children = []

    def __str__(self):
        """Recursively make a string representation of this node."""
        result = self.synset.name
        if len(self.children) > 0:
            children = [rel+str(node) for (rel, node) in self.children
                ]
            result += "(" + ", ".join(children) + ")"
        return result


class Path:
    """A Path object represents a path from one word to another
        through
    synset relations. It consists of a start word, an end word, and a
    start synset which is one of the synsets of the start word.
    It also contains a list of (relation string, synset) pairs which
    are the links in the path.
    """
    def __init__(self, start, end, pathlist):
        self.start_synset = pathlist[0]
        self.start = start # Start word (string)
        self.end = end     # End word (string)
        self.links = []    # List of (relation, Synset) pairs
        for i in range(1, len(pathlist), 2):
            relation = pathlist[i]
            synset = pathlist[i+1]
            self.links.append((relation, synset))

    def length(self):
        """Get the length of this path."""
        return len(self.links)

```

```python
    def __lt__(self, other):
        """Compare this path's length to another. This makes paths sortable."""
        return self.length() < other.length()

    def __str__(self):
        """Show this path as a string; this will be printed with a score."""
        result = "Path "
        result += " [" + self.start + "] "
        result += self.start_synset.name + " "
        for relation, synset in self.links:
            result += relation + " " + synset.name + " "
        result += "[" + self.end + "]"
        return result


def search_paths(searchtree, dest_word):
    """Generate a list of paths from the root of the search tree to the
    given destination word.

    @param searchtree: the SearchTree to search
    @param dest_word: Destination word to search for
    @return: A list (possibly empty) of Path objects
    """
    dest_synsets = wordnet.synsets(dest_word)
    start_nodes = searchtree.children
    paths = []
    for node in start_nodes:
        pathlists = find_pathlists(node, dest_synsets)
        if len(pathlists) > 0:
            shortest = pathlists[0]
            for p in pathlists[1:]:
                if len(p) < len(shortest):
                    shortest = p
            paths.append(Path(searchtree.word, dest_word, shortest))
    return paths


```

```python
def find_pathlists(node, dest_synsets):
    """This is a recursive helper function that is used by search_paths above.
    From a given node, it recursively searches itself and its children for
    synsets that are in the destination synset list until it finds one.
    """
    pathlists = []
    if node.synset in dest_synsets:
        pathlists.append([node.synset])
        return pathlists
    for (relation, child) in node.children:
        child_paths = find_pathlists(child, dest_synsets)
        for path in child_paths:
            pathlists.append([node.synset, relation] + path)
    return pathlists


class Score:
    """A Score object represents a score between a word and a definition.
    It contains a value, as well as all the information that shows how the
    it was made, including the word and defintion, the method name, and other
    details.
    """

    def __init__(self, word, sdef, paths=[], value=0.0, method="None",
                 details="", weights={}):
        """Create a Score object. This constructor method creates Score
        objects for any type of Score, including null scores (no connection)
        path scores (based on a set of paths found between the word and
        definition) or other scores where a value is provided.
        """
        self.word = word
        self.sdef = sdef
        self.paths = paths
        self.value = value
        self.method = method
        self.details = details
        self.weights = weights
        if self.paths:
            self.paths.sort()
            self.value = self.base_value()
            self.method = self.method_str()

    def base_value(self):
        """
        Calculate a number that represents how long the path is
            between
        the word and definition synsets.
        """
        pathlength = sum(self.weighted_relations())
        return 1.0 / (pathlength + 1)

    def weighted_relations(self):
        """
        Make a list of numbers representing how much each relation in
            the
        path contributes to the path length. This uses the weights
            variable
        that is optional passed in when the Score is initialized.
        """
        relations = [rel for (rel, _) in self.paths[0].links]
        results = []
        for rel in relations:
            results.append(self.weights[rel] if rel in self.weights
                else 1)
        return results

    def pathcount_bonus(self):
        """
        Calculate a bonus value score based on the number of paths
            that were
        found between the word and defintion.
        """
        return 0.01 * len(self.paths)

    def pos_bonus(self):
        """
        Calculate a bonus score based on whether the word and
            definition
        have the same part of speech.
        """
        start_synset = self.paths[0].start_synset
        links = self.paths[0].links
        if len(links) > 0:
            _, end_synset = links[-1]
        else:
            end_synset = start_synset
        if start_synset.pos == end_synset.pos:
            return 0.1
        else:
            return 0

    def method_str(self):
        """Make a string describing the "method" of the score."""
        relations = [rel for (rel, _) in self.paths[0].links]
        if len(relations) == 0:
            return "Same␣synset"
        if len(relations) >= 3:
            return str(len(relations)) + "␣or␣more"
        return str(len(relations)) + "␣" + "".join(relations)

    def __lt__(self, other):
        """
        Test whether a score is "less" (i.e. less close, less good)
        than other score. This makes scores sortable.
        """
        return self.value < other.value

    def __str__(self):
        """Make a string showing the score for output."""
        if self.paths:
            result = "Score␣{0:.2f}".format(self.base_value())
```

```
                result += " = 1.0 / ("
                result += str(sum(self.weighted_relations()))
                result += "+1)\n"
                result += "\n".join(["\t" + str(path) for path in self.
                    paths[:1]])
                return result
            else:
                result = "Score {0:.2f}, {1}".format(self.value, self.
                    method)
                if self.details:
                    result += "\n" + self.details
                return result

    def __hash__(self):
        """Make scores hashable (so they can be dictionary keys)."""
        return hash(repr(self))
```