UNIVERSITÀ DEGLI STUDI
DI
MODENA E REGGIO EMILIA

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Risoluzione di ambiguità semantiche per la ricerca di similarità tra frasi

Tesi di Laurea
di
Erika Stefanini

Relatore:
Chiar.mo prof. Paolo Tiberio

Correlatori:
Dott. Federica Mandreoli
Ing. Riccardo Martoglia

Anno Accademico 2002/2003

RINGRAZIAMENTI

Ringrazio la Dottoressa Federica Mandreoli, l'Ing. Riccardo Martoglia e il Professor Paolo Tiberio per la continua disponibilità e assistenza. Un ringraziamento speciale va ai miei genitori, a mia sorella ed ai miei amici che in questi anni mi hanno sempre sostenuto e sopportato.

PAROLE CHIAVE

Ricerca di similarità

Word Sense Disambiguation

Reti semantiche

WordNet

Indice

Indice delle figure

Introduzione

La risoluzione di ambiguità semantiche dei termini (*word sense disambiguation*) è stato un tema di grande interesse fin dagli anni ’50. Il problema di determinare in modo automatico il significato più appropriato di una parola in base al contesto, ossia alla frase, in cui si trova, è infatti un problema non semplice da risolvere, tanto che per decenni è stato studiato nell’ambito dell’intelligenza artificiale, in particolare nel campo della comprensione del linguaggio naturale.
Oltre che in questo campo, il *word sense disambiguation* è stato studiato ed utilizzato in settori quali la traduzione automatica e l’information retrieval.

La tesi è inserita nel progetto EXTRA il cui scopo è affrontare il problema della ricerca di similarità tra frasi nel contesto della traduzione multilingua.
Il progetto EXTRA è uno strumento EBMT (Example Based Machine Translation) il cui compito principale è quello di aiutare il traduttore effettuando la *pretraduzione*: dopo aver inserito documenti già tradotti e il nuovo documento da tradurre, EXTRA costruisce una base di dati di frasi “approvate” e da questa attinge informazioni per suggerire le traduzioni delle frasi del nuovo documento.
Il traduttore può utilizzare ed eventualmente modificare i suggerimenti per la traduzione del nuovo documento. Questa può poi essere a sua volta inserita nella base di dati ed essere utile per una nuova traduzione. Gli algoritmi utilizzati da EXTRA si basano sul concetto di edit distance.

Si è voluto aggiungere al progetto EXTRA una funzionalità di tipo semantico: considerare nei suggerimenti delle traduzioni anche il significato delle parole polisemiche.
Partendo dagli studi compiuti sul word sense disambiguation, è stato realizzato un modulo in grado di determinare il significato più appropriato di sostantivi e verbi presenti in una frase. A tal fine, sono stati implementati algoritmi (uno per i nomi e uno per i verbi) in grado di calcolare un valore che rappresenti la "confidenza" con cui si può dire che il significato x di un termine è migliore degli altri in quel determinato contesto. Utilizzando questi algoritmi insieme al modulo originale di EXTRA, si ottengono suggerimenti per la traduzione che tengono conto del significato dei termini.

La struttura di questa tesi è organizzata in diversi capitoli il cui contenuto verrà ora illustrato.
Il capitolo 1 illustra gli studi fatti finora sul word sense disambiguation.
Il capitolo 2 descrive WordNet, ossia il database lessicale messo a punto presso l'Università di Princeton. In questo capitolo viene descritto cosa è WordNet ed in particolare vengono descritte le relazioni semantiche che stanno alla base della sua struttura.
Il capitolo 3 descrive tutti i passi che sono stati necessari per realizzare il progetto, partendo dallo stemming e dal parsing delle frasi, fino ad arrivare agli algoritmi che calcolano le misure di similarità semantica.
Il capitolo 4 spiega come è stato implementato il software.
Nel capitolo 5, infine vengono illustrati gli esperimenti eseguiti per testare l'efficacia del programma. Vengono inoltre tratte le conclusioni sul lavoro svolto.
Nelle appendici A e B sono riportati rispettivamente un'introduzione al linguaggio Java e il codice Java delle parti più importanti del programma.

Capitolo 1

Word sense disambiguation

1.1 Analisi del word sense disambiguation

Il *word sense disambiguation*[1] coinvolge l'associazione di un termine in un testo con il significato che meglio di altri può essere attribuito a quella parola.
Tale associazione può essere ottenuta eseguendo due passi: prima bisogna recuperare tutti i significati di tutte le parole necessarie alla comprensione del testo, poi bisogna trovare il mezzo con cui assegnare ad ogni parola il significato più appropriato.
Per quanto riguarda il primo passo, molti studi sul word sense disambiguation si basano su significati predefiniti come ad esempio quelli che si trovano nei dizionari, oppure su categorie di parole come i sinonimi. In realtà per ogni significato non esiste una definizione accettata da tutti, è quindi possibile ottenere risultati diversi in relazione al dizionario utilizzato.
Per quanto riguarda invece il secondo passo, l'individuazione del giusto significato di una parola ha una maggiore affidabilità se si tiene conto del contesto in cui è inserita la parola e/o se si utilizzano dati esterni che possano essere utili nell'associare parole e significati. In questo contesto, si possono distinguere tre categorie di lavori sul word sense disambiguation:

- Metodi basati sull'intelligenza artificiale (*AI-based methods*);
- Metodi guidati dalla conoscenza (*Knowledge-driven methods*): metodi in cui il contesto della parola da "disambiguare" è combinato con informazioni provenienti da una sorgente esterna di conoscenze;

- Metodi basati su corpus o dati (*Corpus-based o data-driven methods*): metodi in cui il contesto della parola in esame è combinato con istanze precedentemente disambiguate della parola, provenienti da un corpus.

I primi studi sul word sense disambiguation vennero compiuti nell'ambito della traduzione automatica, i cui obiettivi erano inizialmente modesti; si trattava infatti di tradurre testi tecnici o in ogni caso testi il cui contenuto era ristretto ad un particolare settore. In questo periodo, si ebbe perciò un grande sviluppo di dizionari specializzati, ma ci si rese anche conto della necessità di avere una rappresentazione dei significati delle parole: nacque così, alla fine degli anni '50, il concetto di rete semantica.
Passiamo ora ad illustrare i principali metodi usati per il word sense disambiguation.

1.2 Metodi per il word sense disambiguation

1.2.1 Metodi basati sull'intelligenza artificiale

Nell'ambito dell'intelligenza artificiale il problema della comprensione del linguaggio venne affrontato con l'introduzione delle reti semantiche[2,3,4]. Vennero infatti ideate reti i cui nodi rappresentavano parole e concetti e i cui i legami rappresentano relazioni semantiche tra i nodi. In questo caso, date due parole della rete, l'ambiguità viene risolta perché esiste un solo nodo-concetto legato ad una delle parole che appartiene al cammino più diretto tra le due parole.
Studi successivi presero in considerazione l'utilizzo di frame contenenti informazioni sulle parole, sul loro ruolo e sulle relazioni con altre parole all'interno di singole frasi.
Negli anni '70 furono realizzati i "*spreading activation models*"[5,6,7] in cui i concetti (nodi) presenti in una rete semantica vengono attivati con l'uso e l'attivazione si diffonde ai nodi connessi; ovviamente l'attivazione può provenire da più nodi. Questo modello venne arricchito con la nozione di "inhibition" tra

nodi: l'attivazione di un nodo può comportare l'inibizione di alcuni dei nodi vicini.
Vennero inoltre proposti modelli distribuiti, che avevano il difetto, a differenza dei modelli 'locali' presentati, di richiedere una fase di apprendimento basata su esempi in cui l'ambiguità fosse già stata risolta.
Tutti i sistemi AI-based necessitavano di ampie risorse "trattate a mano": è per questo motivo che essi vennero solitamente applicati a ristrette sezioni del linguaggio.

1.2.2 Metodi guidati dalla conoscenza

I lavori compiuti nell'ambito dell'intelligenza artificiale erano molto interessanti dal punto di vista teorico ma potevano essere utilizzati dal punto di vista pratico solo in determinati ambienti. Un grosso ostacolo per la generalizzazione di tali lavori era dovuta all'enorme quantità di conoscenza richiesta per il word sense disambiguation.
Tutto diventò più semplice negli anni '80 quando divennero disponibili su larga scala risorse lessicali come dizionari, *thesauri* e *corpora*. Furono effettuati tentativi di estrarre in modo automatico informazioni lessicali e semantiche dai machine-readable dictionaries[8,9]; ma l'impresa non andò del tutto a buon termine, visto che l'unica fonte di conoscenza lessicale disponibile su larga scala, ossia WordNet, è stata creata a mano.
Mentre questo tipo di dizionari era pieno di informazioni dal punto di vista lessicale, i *thesauri* tenevano conto anche delle relazioni esistenti tra le parole, come ad esempio la sinonimia. Un *thesaurus* generalmente è organizzato in categorie tali che parole appartenenti alla stessa categoria siano legate dal punto di vista semantico.
Sia i *thesauri* che i dizionari visti prima sono creati dall'uomo perciò non possono essere una perfetta e inconfutabile fonte di informazione sulle relazioni semantiche esistenti tra le parole. In ogni caso, i *thesauri* forniscono un'ampia rete

di associazioni tra parole e un insieme di categorie semantiche di grande valore per tutti i lavori in cui si studia la comprensione della lingua.
A metà degli anni ’80, si cominciarono a costruire a mano basi di conoscenza su larga scala (es. WordNet). Esistono due approcci per la costruzione di dizionari semantici: l’approccio *enumerativo*, in cui i significati sono forniti in modo esplicito, e quello *generativo*, in cui esistono regole usate per estrarre informazioni sul significato.
Fra tutti i vocabolari di tipo enumerativo, WordNet è il più conosciuto ed usato per il word sense disambiguation della lingua inglese (v. capitolo 2). WordNet contiene varie informazioni sia lessicali che di tipo semantico come:

- la definizione dei significati di ogni singola parola in esso contenuta;
- definizione dei synset, ossia insiemi di sinonimi che rappresentano un concetto; i synset sono organizzati in gerarchie come in un thesaurus;
- uso di relazioni semantiche tra le parole come iponimia, iperonimia, antonimia, meronimia.

Anche i dizionari di tipo generativo sono stati utilizzati per il word sense disambiguation[10]; in questo tipo di vocabolari, i significati vengono generati da regole.

1.2.3 Metodi basati su corpus

La ricerca di metodi basati su un corpus per la creazione di dizionari o di “grammatiche” non fece molti passi avanti fino agli anni ’80; fino a tale periodo infatti, l’uso della statistica per l’analisi linguistica avveniva principalmente in campo umanistico e nell’ambito dell’information retrieval. Dagli anni ’80 in poi l’uso dei *corpus* venne rivalutato, anche perché i progressi della tecnologia rendevano possibile la creazione e l’archiviazione di corpora molto grandi a cui potevano essere facilmente applicate leggi di tipo statistico. Ciononostante, l’estrazione di informazioni lessicali dai corpora era ostacolata dalla difficoltà di etichettare manualmente i significati dei termini inclusi in un training corpus (ossia un corpus di riferimento) e dalla rarefazione dei dati (data sparseness).

L'etichettatura manuale di un corpus è infatti un'operazione molto costosa, visto che ad ogni termine incluso nel corpus va associato il proprio significato (alcuni lavori in questo ambito associano alla parola il corrispondente significato di WordNet).
In altri studi si propose l'uso di corpora bilingue per evitare l'operazione di sense-tagging (etichettatura) sulla base della seguente considerazione: significati diversi di una parola vengono tradotti con parole distinte in un'altra lingua (ad esempio, in italiano la parola 'pesca' viene tradotta in inglese come 'peach' nel caso si parli di un frutto e come 'fishing' quando si parla di pescare). Usando un corpus bilingue allineato la traduzione di ogni termine può perciò essere utilizzata per determinare in modo automatico il significato più appropriato. Questo metodo ha comunque i suoi limiti visto che le ambiguità nella lingua destinazione non vengono risolte.
Altri ricercatori proposero addirittura di evitare il sense-tagging dei corpora, ma si notò che questa soluzione non portava a risultati molto affidabili.
Come detto in precedenza, la rarefazione dei dati è un problema importante per tutto ciò che ha a che fare con il word sense disambiguation, sia dal punto di vista dei costi, infatti per rappresentare tutti i possibili significati di una parola bisogna avere a disposizione una quantità enorme di dati, sia dal punto di vista puramente logico, infatti la mancanza di rappresentazione di un significato può determinare un risultato sbagliato quando si applica un algoritmo di word sense disambiguation.

1.3 Problemi da affrontare con il word sense disambiguation

Qualsiasi sia l'approccio scelto per "disambiguare" parole, bisogna affrontare diversi problemi, di cui il più importante è sicuramente il contesto in cui si vengono a trovare le parole. Il contesto è l'unico mezzo per determinare il significato di una parola polisemica.
Il contesto viene usato principalmente seguendo due diversi approcci:

- l'approccio *bag of words* in cui il contesto è dato da un intorno di parole adiacenti alla parola da "disambiguare";

- l'approccio *relational information* in cui il contesto è determinato da relazioni con la parola da "disambiguare" che possono essere la distanza da essa, relazioni di tipo sintattico o semantico, collocazione all'interno della frase, ecc.

Molti lavori nell'ambito del *word sense disambiguation* usano come fonte principale di informazioni il contesto locale (*micro-context*), cioè un piccolo intorno di parole adiacenti alla parola in esame. In questo caso le parole dell'intorno non sono legate da nessun tipo di relazione lessicale o semantica alla parola di cui si vuole determinare il significato. Questo tipo di approccio prende in considerazione intorni che abbiano un raggio massimo di 3 o 4 parole, mentre l'approccio che considera le relazioni tra parole ha bisogno di considerare intorni di qualche decina di parole.
L'approccio bag of words funziona meglio per i nomi che per i verbi, ma in generale è meno efficace dei metodi che usano qualche tipo di relazione tra le parole.

Si è notato che bisogna usare diverse tecniche di *word sense disambiguation* a seconda della categoria sintattica della parola in esame. Infatti i verbi traggono maggiori informazioni sul disambiguation dai loro oggetti piuttosto che dai loro soggetti, gli aggettivi traggono informazioni dai nomi da cui derivano, mentre i nomi sono "disambiguati" meglio dai nomi e aggettivi a loro adiacenti.
Per quanto riguarda invece il raggio dell'intorno del contesto locale, i nomi hanno bisogno di un intorno abbastanza grande mentre i verbi e gli aggettivi necessitano di un intorno piccolo.

Alcuni lavori sul *word sense disambiguation* usano un dominio (*domain*) di significati: questo tipo di approccio attiva solo i significati di una parola rilevanti per il dominio che si sta utilizzando. Questo metodo è poco efficace essendo fonte di errori piuttosto grossolani; se conderiamo la frase *The lawyer stopped at the bar for a drink* nell'ambito giurudico, verrà attivato soltanto il significato di *bar*

relativo a questo dominio (*bar* è l'ordine degli avvocati) e non quello giusto di locale.

Un altro problema che bisogna affrontare con il word sense disambiguation è quello della appropriata determinazione della granularità dei significati. Una granularità come quella che si trova nei dizionari è spesso troppo fine per gli scopi del disambiguation delle parole e comporta una notevole mole di dati da memeorizzare. D'altra parte fare scelte sui significati da introdurre è un compito arduo anche per i lessicografi.

Capitolo 2

Il database lessicale WordNet

Wordnet[11] è un sistema lessicale basato sulle teorie psicolinguistiche della memoria lessicale umana, realizzato presso il Cognitive Science Laboratory della Princeton University da un gruppo di psicologi e linguisti coordinati dal professor Gorge A. Miller.

Attualmente Wordnet è giunto alla alla versione 1.7.1; esso conta 146350 lemmi organizzati in 111223 insiemi di sinonimi (synset)[12]. WordNet non è semplicemente un dizionario di lingua inglese; esso infatti divide l'insieme delle parole in quattro categorie : nomi, verbi, aggettivi e avverbi.

POS	Lemmi	Synsets	Totale coppie parola-significato
Nome	109195	75804	134716
Verbo	11088	13214	24169
Aggettivo	21460	18576	31184
Avverbio	4607	3629	5748
Totale	146350	111223	195817

Fig. 2.1 Conteggio di nomi e synset per ogni categoria di WordNet

Ogni categoria è organizzata in insiemi di sinonimi che rappresentano un determinato concetto; tali insiemi sono inoltre collegati da diversi tipi di relazioni semantiche.
I dizionari tradizionali sono organizzati mediante un ordinamento alfabetico per rendere più semplice e rapida l'individuazione da parte del lettore del termine cercato. Mediante questo approccio, vengono accostati termini che rappresentano concetti totalmente diversi, inoltre per ogni lemma vengono raccolti tutti i significati insieme anche se non hanno niente in comune. Inoltre un comune dizionario è ridondante per quanto riguarda i sinonimi, in quanto lo stesso concetto viene ripetuto più volte.
La caratteristica principale di WordNet è il suo tentativo di organizzare le informazioni lessicali delle parole in base al significato delle parole stesse e non alla loro forma (lemma).

2.1 La matrice lessicale

Ogni parola è l'associazione tra la sua forma (il modo in cui viene scritta) e il significato che essa esprime. D'ora in poi indicheremo con "word form" o lemma il modo in cui viene scritta la parola e con "word meaning" o significato il concetto che essa esprime.
La corrispondenza tra il lemma delle parole ed il significato che esprimono viene rappresentato nella matrice lessicale, in cui le righe riportano i "word meaning" e la colonne le "word form" (vedi Fig. 2.2). Un elemento $E_{i,j} = (M_i , F_j)$ della matrice rappresenta una definizione: il lemma F_j è usato per esprimere il significato dato da M_i .
Se ci sono due elementi nella stessa riga significa che le due word form sono sinonimi, mentre se ci sono due elementi nella stessa colonna significa che la word form è polisemica (cioè ha più significati).
La corrispondenza tra lemmi e significati è di tipo molti a molti, infatti esistono lemmi che hanno più significati e significati che possono essere espressi da più lemmi.

Word Meanings	Word Forms				
	F_1	F_2	F_3	…	F_n
M_1	$E_{1,1}$	$E_{1,2}$			
M_2		$E_{2,2}$			
M_3			$E_{3,3}$		
…				…	
M_m					$E_{m,n}$

Fig. 2.2 Matrice lessicale

Un concetto può essere rappresentato essenzialmente in due modi: o attraverso una definizione come quelle presenti nei dizionari cartacei o attraverso un insieme di sinonimi del concetto stesso. Nel primo caso, può non esserci concordanza sulla definizione da dare. La seconda scelta si adatta maggiormente al caso in cui il lettore conosca già il termine e abbia solo bisogno di identificare il significato più adatto al contesto in cui la parola è inclusa. WordNet adotta la seconda modalità.

2.2 Relazioni tra lemmi e significati

Wordnet è organizzato mediante relazioni di tipo semantico ossia relazioni tra insiemi di sinonimi e relazioni di tipo lessicale. Queste ultime mettono in corrispondenza gli aggettivi di tipo “relazionale” al nome a cui sono legati e gli avverbi con gli aggettivi da cui derivano.

Le relazioni semantiche sono per la maggior parte di tipo simmetrico: se esiste una relazione R tra il synset { x_1 , x_2 , …. } e il synset { y_1 , y_2 , …. }, allora esiste anche una relazione R’ tra { y_1 , y_2 , …. } e { x_1 , x_2 , …. }.

Per rappresentare le relazioni tra parole appartenenti a synset diversi si usano i puntatori. I seguenti tipi di puntatore sono usati per indicare relazioni di tipo lessicale: “antonym”, “pertainym”, “participle”, “also see”; i restanti tipi di

puntatore sono generalmente usati per rappresentare relazioni di tipo semantico. Sebbene ci siano molti tipi di puntatori, solo alcuni tipi di relazione sono permessi all'interno di ogni categoria.

In figura 2.3 sono indicati i simboli dei puntatori usati da WordNet divisi per categoria sintattica; in figura 2.4 sono riportate le relazioni di tipo simmetrico e le corrispondenti relazioni "inverse".

Nomi	Verbi	Aggettivi	Avverbi
! antonym	! antonym	! antonym	! antonym
@ hypernym	@ hypernym	& similar to	\ derived from adjective
~ hyponym	~ hyponym	< participle of verb	
#m member holonym	* entailment	\ pertainym	
#s substance holonym	> cause	= attribute	
#p part holonym	^ also see	^ also see	
%m member meronym	$ verb group		
%s substance meronym			
%p part meronym			
= attribute			

Fig. 2.3 Simboli dei puntatori divisi per categoria sintattica

Pointer	Reflect
Antonym	Antonym
Hyponym	Hypernym
Hypernym	Hyponym
Holonym	Meronym
Meronym	Holonym
Similar to	Similar to
Attribute	Attribute
Verb Group	Verb Group

Fig. 2.4 Relazioni di tipo simmetrico e loro inverse

All'interno del database di Wordnet, una entry per una parola contiene i puntatori

alle entry di altre parole. Nell'esempio raffigurato in figura 2.5 viene mostrata una parte dei puntatori del synset "apple": il simbolo ~ indica che *apple* è un iponimo di *edible fruit*, mentre il simbolo %m indica che *apple* è "member meronym" di *apple tree* (*apple* è membro di *apple tree*); viene inoltre riportata la relazione inversa a quella di iponimia tra i termini apple e edible fruit (indicata con il simbolo @).

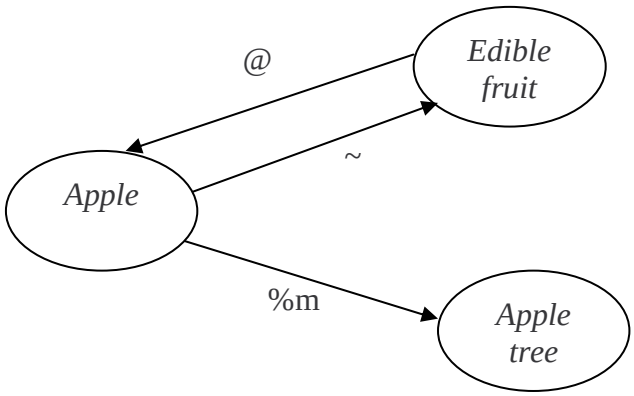


Fig. 2.5 Esempio di relazioni semantiche

Procediamo ora ad illustrare le principali relazioni semantiche usate da WordNet.

2.2.1 Sinonimia

La relazione più importante in Wordnet è senz'altro la sinonimia, visto che essa è alla base della costruzione della matrice lessicale. Secondo una definizione attribuita a Leibniz, due espressioni sono sinonime se la sostituzione di una per l'altra non cambia il valore della frase in cui avviene la sostituzione. Da tale definizione, risulta chiaro che i veri sinonimi sono molto rari. Una definizione meno forte esprime la sinonimia nel seguente modo: due espressioni sono sinonime in un contesto C se la sostituzione di una per l'altra in C non modifica il valore della frase.

Questa definizione di sinonimia rende necessaria la divisione in nomi, verbi,

aggettivi e avverbi di WordNet in modo che parole appartenenti a categorie diverse non possano essere sinonime visto che non possono essere intercambiabili.

2.2.2 Antonimia

La relazione di antonimia associa ad un termine il suo contrario . L'antonimo di una parola x, a volte è –x, ma non sempre: ad esempio *white* e *black* sono antonimi, ma dire che un oggetto non è bianco, non significa che sia necessariamente nero.

Questo tipo di relazione può essere usato per coppie di termini appartenenti a tutte le categorie sintattiche in cui è suddiviso Wordnet.

L'antonimia non è una relazione semantica tra word meanings ma è una relazione lessicale tra lemmi. Questa affermazione può essere rappresentata con un esempio: i significati {rise, ascend} e {fall, descend} sono concettualmente opposti ma non sono antonimi; rise e fall sono antonimi così come ascend e descend ma non si può dire che lo siano rise e descend o ascend e fall.

Perciò risulta chiaro che bisogna distinguere tra relazioni semantiche tra word forms e relazioni semantiche tra word meaning.

2.2.3 Iponimia/iperonimia

Un concetto rappresentato dal synset {x, x', …} è un iponimo del concetto rappresentato dal synset {y, y', …} se può essere costruita una frase del tipo *An x is a (kind of) y* .

Questo tipo di relazione è permesso solo per le categorie dei nomi e dei verbi; nel caso dei verbi, la realzione di iponimia viene chiamata "troponimia".

La relazione di iponimia/iperonimia , chiamata anche relazione ISA, è transitiva e simmetrica; essa genera una struttura gerarchica semantica simile alla gerarchia di specializzazione/generalizzazione presente nei modelli relazionali. La radice di tale gerarchia è occupata dai concetti più generali, mentre al livello delle foglie ci sono i concetti specializzati.

Come nei modelli relazionali e nei modelli ad oggetti, ogni iponimo eredita tutte

le caratteristiche del concetto più generale e al più viene aggiunta una caratteristica che distingue l'iponimo dal concetto al livello superiore e dagli altri iponimi di quel concetto. Ad esempio la parola *apple* (vedi fig. 2.6) eredita tutte le caratteristiche del suo iperonimo *fruit* ma si distingue da tutti gli altri tipi di frutto per la sua forma, il colore, il gusto, ecc.

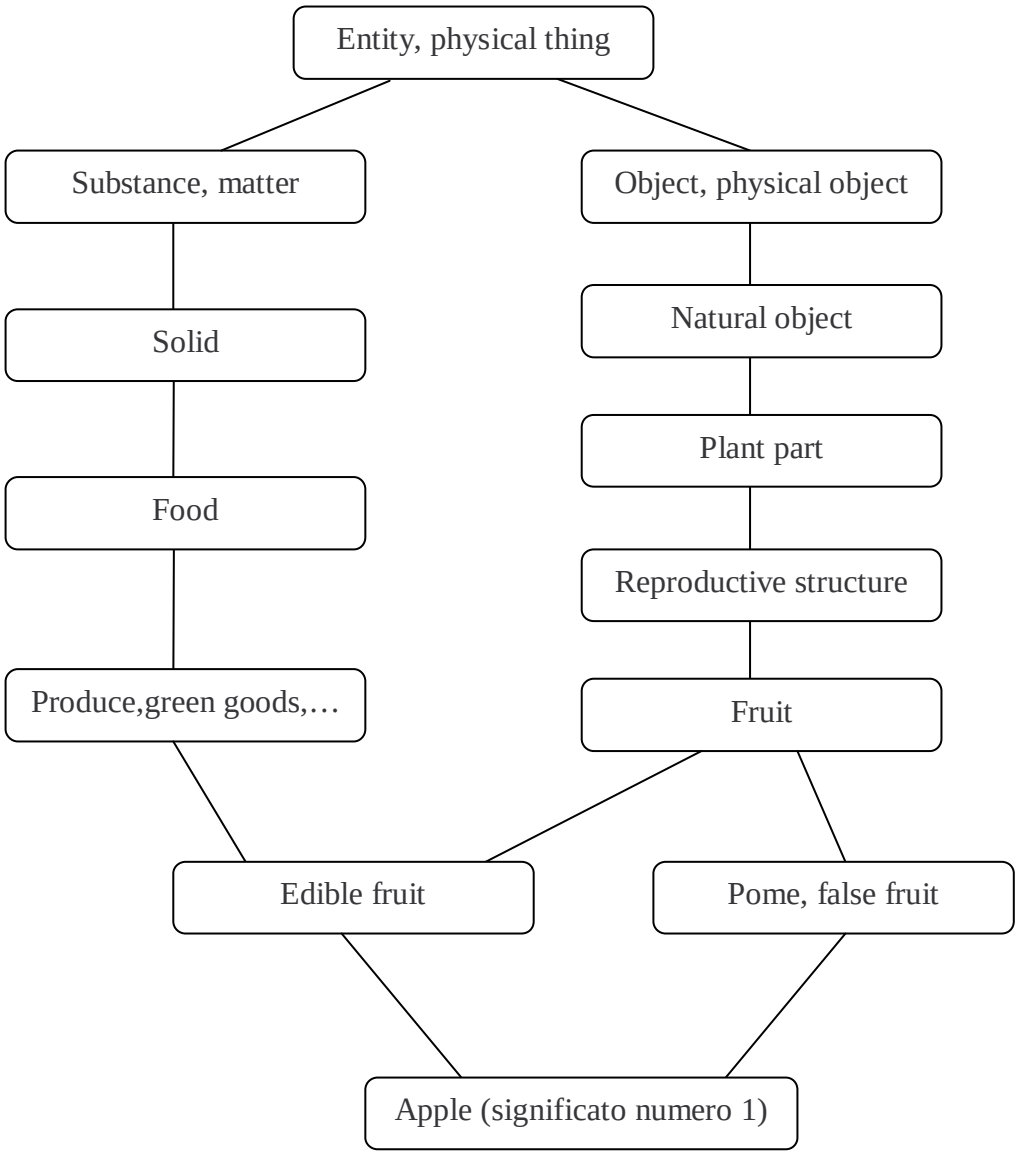


Fig. 2.6 Esempio di gerarchia ISA (applicato al significato 1 di *apple*)

Come detto all'inizio del paragrafo, nel caso dei verbi non si parla di iponimia ma di "troponimia" (troponymy); tale termine è stato introdotto da FellBaum e Miller[13] . La frase *An x is a y* usata per testare l'iponimia tra nomi non è adatta per i verbi, perché viene implicitamente richiesto che x e y siano nomi. Nel caso delle forme verbali si dice che V_1 è un "troponimo" di V_2 se risulta vera l'espressione *To V_1 is to V_2 in some particular manner.*

Si può anche dire che la "troponimia" è un particolare tipo di entailment (vedi paragrafi 2.2.5 e 2.3.2) , in quanto ogni "troponimo" V_1 di un verbo più generale V_2 implica (entails) anche V_2.

2.2.4 Meronimia (relazione part of)

Un altro tipo di relazione semantica importante per i lessicografi è la relazione di meronimia/olonimia, detta anche relazione parte-tutto; essa si applica solo alla categoria dei nomi.

Un concetto rappresentato da un synset {x, x', …} è un meronimo del concetto {y, y', …} se può essere costruita una frase del tipo: *A y has an x (as a part)* oppure *An x is a part of y* .

La relazione di meronimia è transitiva e asimmetrica; essa può essere usata per costruire una gerarchia (bisogna considerare però che un meronimo può avere più olonimi). I meronimi possono essere usati per distinguere un concetto da un altro all'interno della gerarchia di iponimia.

In WordNet sono definiti 3 tipi di aggregazione:

part : x è part meronym di y se x è un componente o una parte di y (es: *wheel* è part meronym di *bike*);

member : x è member meronym di y se x è un membro di y;

substance : x è substance meronym di y se x è il materiale con cui è fatto y.

La relazione di substance meronimy ha in generale i suoi limiti: ogni oggetto concreto è composto di "atomi", perciò "atomo" sarebbe meronimo di ogni oggetto tangibile e non si potrebbe più distinguere tra una categoria di oggetti ed un'altra. Per questo motivo il "sezionamento" di un oggetto termina quando le

parti che lo specializzano non servono più a distinguerlo dagli altri oggetti.

2.2.5 Implicazione (entailment)

Questa relazione può essere utilizzata solo per i verbi.

Un verbo x implica y se x non può essere fatto senza che y sia o sia stato fatto. Questo tipo di relazione non è simmetrico, infatti se x implica y , non è vero il contrario, tranne nel caso in cui i due verbi siano sinonimi.

La relazione di entailment è simile a quella di meronimia nei nomi, ma la meronimia si applica meglio ai nomi che ai verbi.

Esempio : *eat* / *chew*

L'entailment è spiegato in modo più dettagliato nel paragrafo 2.3.2.

2.2.6 Relazione causale (cause to)

Anche questa relazione si applica solo alla categoria dei verbi.

La relazione causale lega due verbi, uno che produce una causa, che chiameremo per motivi di brevità C, e uno che "riceve" un effetto, che chiameremo E. A differenza di altre relazioni codificate in WordNet, il soggetto del verbo C solitamente ha un referente che è diverso dal soggetto del verbo E; il soggetto del verbo E deve essere un oggetto per il verbo C, il quale deve essere dunque transitivo.

Esempio : *show* / *see* ; in questo caso *show* è il verbo C, mentre *see* è il verbo E

2.2.7 Similarità (similar to)

La relazione di similarità è tipica della categoria degli aggettivi e non è da confondersi con la similarità semantica che verrà introdotta nei prossimi capitoli.

Questo tipo di similarità è simile alla sinonimia.

WordNet divide gli aggettivi in due classi principali: descrittivi e relazionali. Un aggettivo descrittivo è un aggettivo che attribuisce un valore di un attributo ad un

nome. Ad esempio, dire *The package is heavy* presuppone che ci sia un attributo *weight* tale che weight(package) = heavy.

I synset riguardanti gli aggettivi descrittivi sono organizzati in cluster di aggettivi. Al centro di questi cluster c'è un aggettivo a cui gli altri componenti sono legati da similarità. Gli aggettivi che si trovano al centro dei cluster sono legati mediante la relazione di antonimia ad altri cluster. Gross, Fischer e Miller[14] distinguono tra antonimi diretti come heavy/light e antonimi indiretti come heavy/weightless. In questo modo gli aggettivi che non hanno antonimi diretti hanno però antonimi indiretti (questi ultimi vengono ereditati attraverso la relazione di similarità). In figura 2.7 è visualizzato un esempio di cluster contrapposti; gli aggettivi *wet* e *dry* costituiscono una coppia di antonimi diretti, l'aggettivo *watery* non ha un antonimo diretto ma ne ha uno indiretto, ossia *dry*.

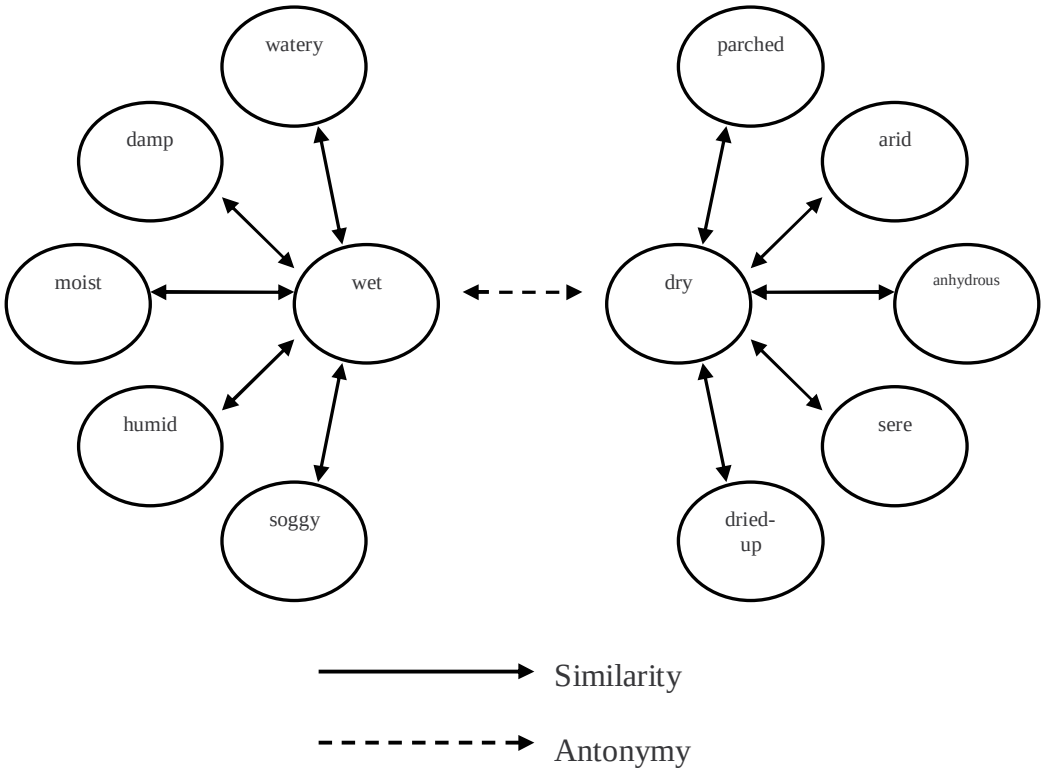


Fig. 2.7 Esempio di struttura degli aggettivi

2.2.8 Relazione di pertinenza (pertainym)

La relazione di pertinenza può essere applicata alla categoria degli aggettivi (la relazione inversa “derived from adjective” viene applicata agli avverbi).
Gli aggettivi che rimangono fuori dall’organizzazione per cluster contrapposti sono gli aggettivi relazionali. Tali aggettivi sono definiti da frasi del tipo “di o pertinenti a” e non possiedono antonimi. Un aggettivo di pertinenza può essere in relazione con un nome o con un altro aggettivo di questo tipo.

Esempio: l’aggettivo *dental* è in relazione di pertinenza con il sostantivo *tooth*

2.2.9 Relazione participiale (participle)

Anche questa relazione è tipica della categoria degli aggettivi; un aggettivo di questo tipo è un aggettivo che deriva da un verbo.

Esempio : l’aggettivo *burned* deriva dal verbo *burn*

2.2.10 Attributo (attribute)

Questa relazione può essere applicata alle categorie dei nomi e degli aggettivi
Attributo è il nome per cui uno o più aggettivi esprimono un valore. Il nome *weight* è un attributo a cui gli aggettivi light e heavy danno un valore.

2.2.11 Coordinato

Non è un tipo di relazione semantica. Due nomi o due verbi sono coordinati se hanno lo stesso ipernimo, cioè se sono la specializzazione dello stesso concetto.

2.3 Organizzazione delle categorie sintattiche

Come è stato detto in precedenza, WordNet divide le parole in categorie sintattiche. Questo tipo di suddivisione genera un po’ di ridondanza in quanto alcuni termini sono inclusi in più categorie, per esempio la parola *cream* è inclusa

nella categoria dei sostantivi, in quella dei verbi e in quella degli aggettivi. Si ha però un grande vantaggio: quello di rendere evidenti le differenze di organizzazione semantica.
WordNet organizza in modo diverso le categorie:

- i nomi sono organizzati mediante gerarchie di generalizzazione/specializzazione basate sulla relazione semantica di iperonimia/iponimia;
- i verbi sono organizzati tramite varie relazioni di implicazione;
- infine aggettivi e avverbi sono organizzati in iperspazi N-dimensionali.

Ora tratteremo solamente l'organizzazione relativa a nomi e verbi perché sono le uniche categorie usate per lo sviluppo di questa tesi.

2.3.1 Organizzazione dei nomi

Wordnet contiene circa 110000 nomi organizzati in circa 75000 synsets; tali valori sono relativi all'ultima versione e sono indicativi in quanto WordNet continua a crescere. In termini di copertura, WordNet raggiunge quella di un buon dizionario tascabile. Ciò che però lo rende diverso da un dizionario cartaceo è la sua organizzazione interna.
Quando si usa un normale vocabolario di lingua inglese, in esso sono contenute, per ogni termine, informazioni quali lo spelling, la pronuncia, l'etimologia, l'analisi grammaticale, sinonimi e contrari, la definizione dei vari significati ed eventuali esempi d'uso. WordNet contiene solo una parte di queste informazioni, ma contiene altri tipi di informazioni che un comune dizionario non possiede.
Ad esempio, prendiamo la definizione di *tree* nel significato di pianta: un comune vocabolario potrebbe includere una definizione del tipo : *a plant that is large , woody, perennial and as a distinct trunk*. Tale descrizione sembra esaurente, ma non dice che un albero ha delle radici, delle foglie, che le sue cellule contengono cellulosa, ecc.
In realtà, queste informazioni sono presenti nella definizione di *plant* (che è anche ipernimo di *tree*), ma non viene detto da nessuna parte quale dei significati di

plant bisogna andare a controllare, ossia non viene specificato quale sia il significato di *plant* che è iperonimo di *tree*.

Ogni vocabolario contiene dei "circoli viziosi", ossia può capitare che per definire la parola X si usi Y e viceversa. I lessicografi che hanno progettato WordNet hanno tentato di dare alla memoria semantica dei nomi una forma ad albero.
Questo albero può essere ricostruito a partire dal cammino effettuato dagli ipernimi; esempio: oak @→tree @→woody plant @→vascular plant @→plant @→organism … dove si ricorda che @→ è il simbolo che indica la relazione di iperonimia. Questa gerarchia, che è limitata in profondità (può comunque raggiungere 16 livelli di profondità), può essere percorsa sia partendo dai concetti più generali andando verso quelli più specializzati sia nella direzione opposta, poiché in WordNet viene codificata sia la relazione di iperonimia sia la sua relazione inversa, l'iponimia.

Questo tipo di struttura ad albero è molto utile perchè le informazioni comuni a più termini vengono memorizzate una volta sola; tale struttura possiede tutte le proprietà di ereditarietà tipiche delle gerarchie di specializzazione.
Oltre ai puntatori per le relazioni di iperonimia e iponimia, ogni synset relativo ad un nome contiene anche i puntatori per gli altri tipi di relazione validi per tale categoria (vedi paragrafo 2.1) . La definizione di un significato di un nome viene data da un iperonimo e da altre caratteristiche (es: un meronimo, un omonimo, oppure un antonimo) che servono a distinguerlo da altri termini e/o da altri significati.

Si può dire che la struttura dei nomi di WordNet segue i principi che governano la memoria lessicale umana, infatti quando ci si chiede di dare la definizione di un nome, generalmente si tende a dare o un sinonimo della parola stessa o un concetto più generale, in alcuni casi si tende a dare anche altre caratteristiche che lo possano qualificare; se si tratta di un oggetto tangibile si tende ad indicare il materiale di cui è composto o le parti che lo compongono.

Da quanto è stato detto finora, sembrerebbe che la gerarchia dei nomi abbia un'unica radice; in realtà se così fosse il concetto contenuto nella radice sarebbe così generico da fornire poche informazioni a livello semantico.
L'alternativa, adottata da WordNet, è quella di ripartire i nomi in modo da selezionare un numero limitato di concetti generici che verranno trattati ognuno come il capostipite di una gerarchia a sé. La lista dei concetti capostipiti selezionati da WordNet è riportata in figura 2.8.

{ entity, physical thing }
{ psycological feature }
{ abstraction }
{ state }
{ event }
{ act, human action, human activity }
{ group, grouping }
{ possession }
{ phenomenon }

Fig. 2.8 Lista dei concetti capostipite usati da WordNet

2.3.2 Organizzazione dei verbi

Sebbene ogni frase di lingua inglese, per essere corretta dal punto di vista grammaticale, debba contenere almeno un verbo, i verbi inglesi sono in numero decisamente inferiore rispetto ai nomi. In compenso, i verbi hanno una polisemia molto più alta rispetto a quella dei nomi. Le informazioni riguardanti la polisemia di nomi e verbi inclusi in WordNet sono raccolte nelle tabelle di figura 2.9.

POS	Parole e significati monosemici	Parole polisemiche	Significati polisemici
Nome	94685	14510	40002
Verbo	5920	5168	18221
Aggettivo	15981	5479	15175
Avverbio	3820	787	1900
Totale	120406	25944	75298

POS	Polisemia media includendo parole monosemiche	Polisemia media escludendo parole monosemiche
Nome	1.23	2.75
Verbo	2.17	3.52
Aggettivo	1.45	2.76
Avverbio	1.24	2.41

Fig. 2.9 Informazioni sulla polisemia dei termini contenuti in WordNet (versione 1.7.1)

Un'altra caratteristica dei verbi è quella di cambiare significato in base al nome che li accompagna; nelle frasi *I have a Ferrari* e *I have a headache* il verbo *have* assume due significati molto diversi.

I verbi di WordNet sono divisi sulla base di criteri semantici in 15 gruppi, ognuno memorizzato in un file. Tutti questi gruppi, tranne uno, corrispondono a ciò che i linguisti chiamano domini semantici e ogni verbo contenuto in questi insiemi descrive eventi o azioni; il restante insieme non costituisce un dominio semantico e contiene verbi che descrivono stati.
In figura 2.10 viene visualizzata la lista di questi gruppi e una breve descrizione di ciò che contengono.

Nome del file	Descrizione(di cosa trattano i verbi)
verb.body	vestire e cura del corpo
verb.change	cambiamento di dimensione, temperatura, intensità
verb.cognition	pensare, giudicare, analizzare
verb.communication	parlare, domandare, cantare, ecc.
verb.competition	combattimento, attività atletiche, ecc.
verb.consumption	mangiare e bere
verb.contact	toccare, colpire, tirare, ecc.
verb.creation	cucire, cucinare, dipingere, attività creative manuali
verb.emotion	sentimenti
verb.motion	camminare, volare, nuotare, ecc.
verb.perception	vedere, ascoltare, percepire, ecc
verb.possession	comprare, vendere, possedere, trasferire
verb.social	eventi ed attività sociali e politiche
verb.stative	essere, avere, relazioni spaziali
verb.weather	pioggia, neve, meteorologia in genere

Fig. 2.10 Lista dei gruppi in cui sono divisi i verbi di WordNet

Il tipo di struttura usato per i nomi non può essere usato per i verbi. Questi ultimi infatti sono organizzati tramite gerarchie più complesse basate su una serie di relazioni di entailment (rappresentate in fig. 2.11). Queste strutture sono meno profonde ma più “ramificate” rispetto a quelle dei nomi.

Alcune attività possono essere spezzate in altre attività ordinate in modo sequenziale. Fare questo dal punto di vista lessicale significa suddividere l’azione descritta da un verbo in più azioni che possono essere ordinate dal punto di vista temporale oppure no.
Ad esempio la relazione che intercorre tra i verbi *ride* e *drive* è diversa da quella esistente tra *snore* e *sleep*. Nel primo caso, infatti le due attività sono connesse tra

di loro visto che quando si guida un veicolo, necessariamente lo si conduce anche; queste due attività non sono successive ma contemporanee. Nel secondo caso, l'attività del russare (s*nore*) può far parte dell'attività del dormire (*sleep*): le due attività sono almeno parzialmente contemporanee. Perciò la differenza tra le coppie ride/drive e snore/sleep è dovuta alle relazioni temporali che intercorrono tra i membri di ciascuna coppia : le attività possono essere simultanee (prima coppia) o essere incluse una nell'altra (seconda coppia).
Le coppie di verbi viste oltre ad essere legate dalla relazione di entailment possono avere la caratteristica dell'inclusione temporale (temporal inclusion): uno degli elementi della coppia include temporalmente l'altro.

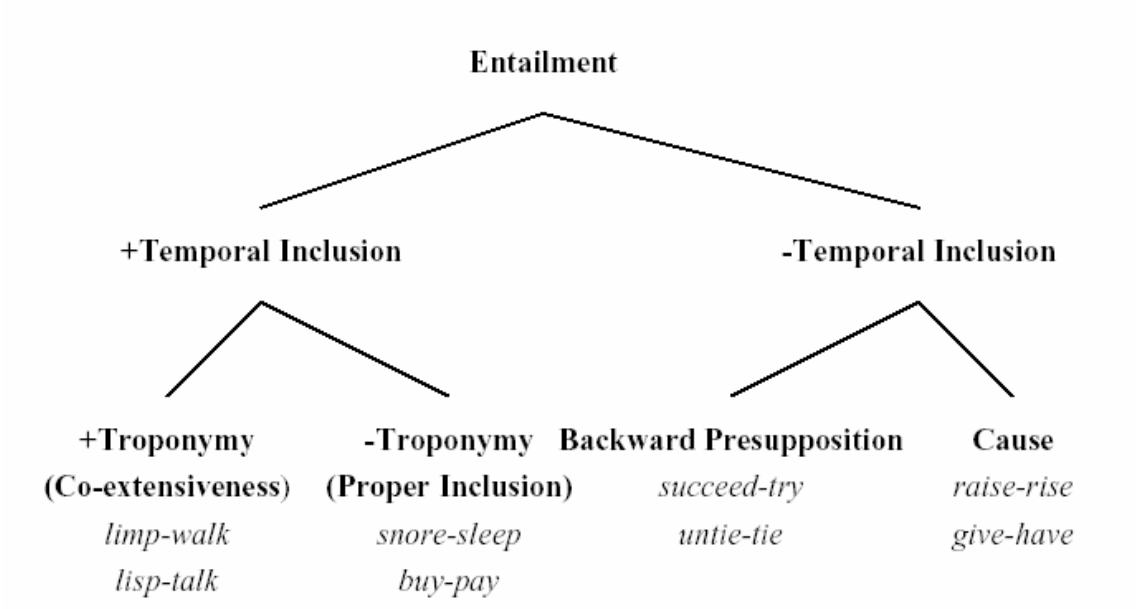


Fig. 2.11 Tipi di entailment

Come già spiegato nel paragrafo 2.2.3, per i verbi viene usata la relazione di troponymy al posto dell'iponimia. Ricordo che il verbo V_1 è in relazione di troponymy con V_2 se si può affermare che V_1 è V_2 in qualche particolare modo.

La relazione di troponymy è un particolare tipo di entailment, infatti considerando la coppia di verbi *limp*/*walk* si può dire che zoppicare (*limp*) è un modo particolare di camminare (*walk*).
Le azioni che si riferiscono ad un troponym sono sempre di tipo simultaneo (totale o parziale): considerando l'esempio precedente, una persona cammina in ogni istante in cui zoppica.

Consideriamo ora l'entailment che non possiede la caratteristica dell'inclusione temporale.
Molti verbi con significati opposti condividono una relazione di entailment con lo stesso verbo. Ad esempio, i verbi *fail* e *succeed* sono entrambi in relazione con *try*, *win* e *lose* lo sono con *play*. Queste coppie di verbi sono connesse da un tipo di entailment detto di backward presupposition .
La relazione causale è un tipo di entailment senza inclusione temporale, visto che i verbi connessi da questo tipo di legame lo sono dal punto di vista logico, non c'è un vincolo temporale.

Capitolo 3

Algoritmi per il *word sense disambiguation*

3.1 Introduzione

Lo scopo della presente tesi è stato la realizzazione di un modulo indipendente per la risoluzione di ambiguità semantiche di termini appartenenti ad una frase in lingua inglese; tale modulo verrà poi usato all'interno del progetto EXTRA[15] per la ricerca di similarità tra frasi. In questo modo si combinano due aspetti molto importanti: la similarità sintattica e quella semantica.

EXTRA è uno strumento EBMT (Example Based Machine Translation) che viene utilizzato nell'ambito della traduzione multilingua.

Esso affronta il problema della ricerca di similarità tra frasi dal punto di vista sintattico, basando i propri algoritmi sul concetto di edit distance, ossia il numero minimo di operazioni necessarie per trasformare una stringa in un'altra (ad esempio, le parole *surgery* e *survey* hanno edit distance pari a 2 e quindi una forte similarità sintattica).

Mediante il modulo realizzato nella presente tesi si è voluto aggiungere al progetto EXTRA uno strumento in grado di determinare, attraverso il calcolo della similarità semantica delle parole, la confidenza con cui si può individuare il significato del termine preso in considerazione.

Nel modulo realizzato si tiene conto solo dei nomi e dei verbi compresi nelle frasi da esaminare; per tali categorie sono stati implementati algoritmi che determinano quale sia, tra i vari significati di WordNet, quello più adatto al vocabolo in esame.

In particolare, l'algoritmo riguardante i nomi prevede la possibilità di considerare, oltre ai nomi presenti nella frase che si sta valutando, anche i nomi presenti in una o più frasi precedenti e successive.

3.2 Passi fondamentali

Prima di procedere al calcolo della similarità semantica, ogni frase sorgente subisce una trasformazione che è fatta di 3 passi: parsing, stemming, combinazione dei due passi precedenti (vedi fig. 3.1).

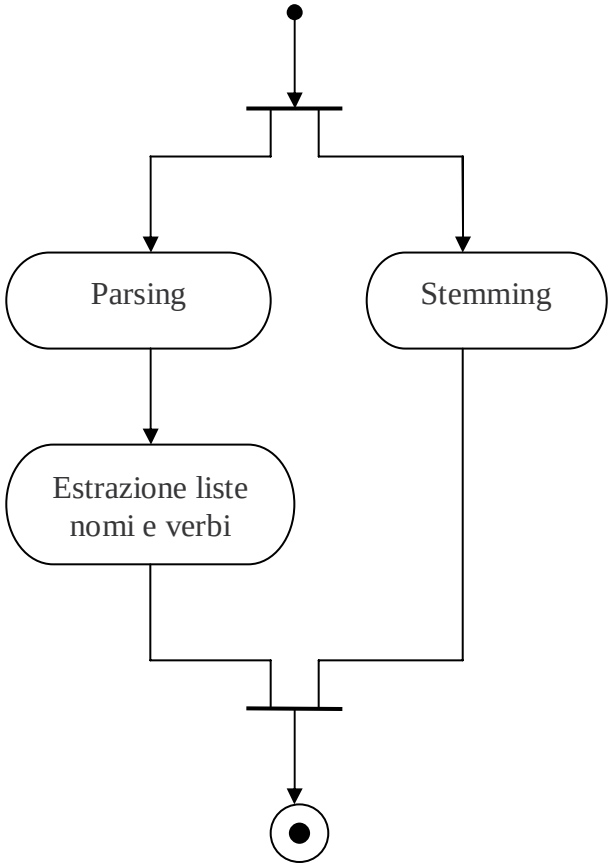


Fig.3.1 Activity diagram delle trasformazioni subite dalla frase sorgente

Vediamo ora di illustrare in cosa consistono queste trasformazioni.

Lo stemming è un processo che comporta l'eliminazione di parole molto comuni (stopwords) come articoli e preposizioni, e la normalizzazione delle altre parole: i nomi vengono portati al singolare, i verbi all'infinito, ecc. Per lo stemming è stato utilizzato un package realizzato da F. Gavioli[16] per la lingua inglese.

Il parsing effettua l'analisi grammaticale della frase: ad ogni parola associa un tag che ne indica il tipo: nome, verbo, aggettivo, avverbio, articolo, preposizione, ecc. Ad esempio la frase "The white cat is hunting the mouse" diventa "The/DT white/JJ cat/NN is/VBZ hunting/VBG the/DT mouse/NN".
Per effettuare questo tipo di operazione, è stato utilizzata la versione 1.0 del tagger Monty Tagger (scaricabile dal sito http://web.media.mit.edu/~hugo/montytagger). Questo tagger utilizza il set di tag definiti dal Penn Treebank Project[17,18,19]; di seguito è riportata la lista completa dei tag e il loro significato.

Simbolo	Significato
CC	Congiunzione
CD	Numero cardinale
DT	Articolo (anche *every*, *some*, ecc)
EX	There "esistenziale" (quando viene usato come soggetto)
FW	Parola straniera
IN	Preposizione
JJ	Aggettivo
JJR	Aggettivo comparativo
JJS	Aggettivo superlativo
LS	Lettere e numeri
MD	Verbo modale (can, could, would, ecc)
NN	Nome singolare
NNS	Nome plurale
NNP	Nome proprio singolare
NNPS	Nome proprio plurale
PDT	Parole tipo *all*, *many*, *both*, *such*
POS	Indica il genitivo sassone 's
PRP	Pronome personale
PRP$	Pronome possessivo
RB	Avverbio
RBR	Avverbio comparativo
RBS	Avverbio superlativo
RP	Parola monosillabica tipo *off*, *up*, ecc.
SYM	Simbolo

TO	To
UH	Esclamazione
VB	Verbo, forma base
VBD	Verbo, passato
VBG	Verbo, gerundio o participio presente
VBN	Verbo, participio passato
VBP	Verbo, presente, non terza persona singolare
VBZ	Verbo, presente, terza persona singolare
WDT	Wh-word che precede un nome
WP	Wh-pronome (inizia con wh)
WP$	Wh-pronome possessivo
WRB	Wh-avverbio

Grazie al parsing è possibile analizzare i termini contenuti nella frase al fine di estrarre la lista dei nomi e quella dei verbi compresi nella frase.
Dunque la frase dell'esempio precedente subisce le seguenti trasformazioni:

Frase originale:	The white cat is hunting the mouse.
Frase all'uscita dello stemmer:	white cat be hunt mouse
Frase all'uscita del parser:	The/DT white/JJ cat/NN is/VBZ hunting/VBG the/DT mouse/NN

La lista dei nomi comprende *cat* e *mouse*, quella dei verbi *be* e *hunt*.

Dopo aver subito queste trasformazioni, la frase è ora pronta per essere elaborata dal punto di vista semantico.
Usando le liste dei nomi e dei verbi si procederà al calcolo della similarità semantica mediante un paio di algoritmi. Queste procedure calcolano il significato più appropriato del nome o del verbo presente nella frase. Con i risultati ottenuti, ogni termine contenuto nel periodo viene sostituito dal corrispondente codice identificativo del significato che è apparso essere il più adatto per tale parola. Tale codice è univoco e viene assegnato all'interno di WordNet ad ogni significato di ogni parola.
Considerando sempre la frase utilizzata nell'esempio, si ottiene un risultato di questo tipo:

Frase “stemmizzata”: white cat be hunt mouse

Frase con i codici di WordNet:

white *1788952* *2058045* *903354* *1993014*

Come si può notare il termine *white* non viene sostituito da nessun codice perchè è un aggettivo e perciò non viene considerato dagli algoritmi di word sense disambiguation. Nel caso in cui una parola non venga trovata nel dizionario di WordNet non viene inserito nessun codice ma viene lasciata la parola originale. Per esempio, se nella frase precedente non venisse trovato il verbo hunt si avrebbe questo tipo di risultato: white *1788952* *2058045* hunt *1993014*.

Per quanto riguarda l’insieme delle delle parole di riferimento, è stata vagliata la possibilità di calcolare, fissando un parametro, la similarità semantica dei nomi della frase, tenendo conto delle frasi adiacenti ad essa nel testo che si sta analizzando. Ciò viene fatto perché nelle frasi adiacenti potrebbero esserci parole utili per il word sense disambiguation dei termini contenuti nella frase sorgente. Consideriamo il caso in cui nella frase sia presente la parola *mouse*: se nelle frasi adiacenti si parla di animali, allora *mouse* è da intendersi col significato di roditore, se invece si parla di computer allora *mouse* è da intendersi col significato di dispositivo elettronico. Questo procedimento viene applicato solo ai nomi perché nel caso dei verbi non è molto significativo.

In particolare alla lista dei nomi della frase vanno aggiunti anche i nomi appartenenti alle frasi adiacenti. Mediante questa nuova lista si procederà al calcolo della misura di similarità semantica.

Nella figura 3.2 sono illustrati i passi fondamentali eseguiti per la risoluzione di ambiguità. L’attività di elaborazione viene eseguita per ogni frase contenuta nel documento che si vuole analizzare.

La generazione dei codici è relativa alla frase sorgente.

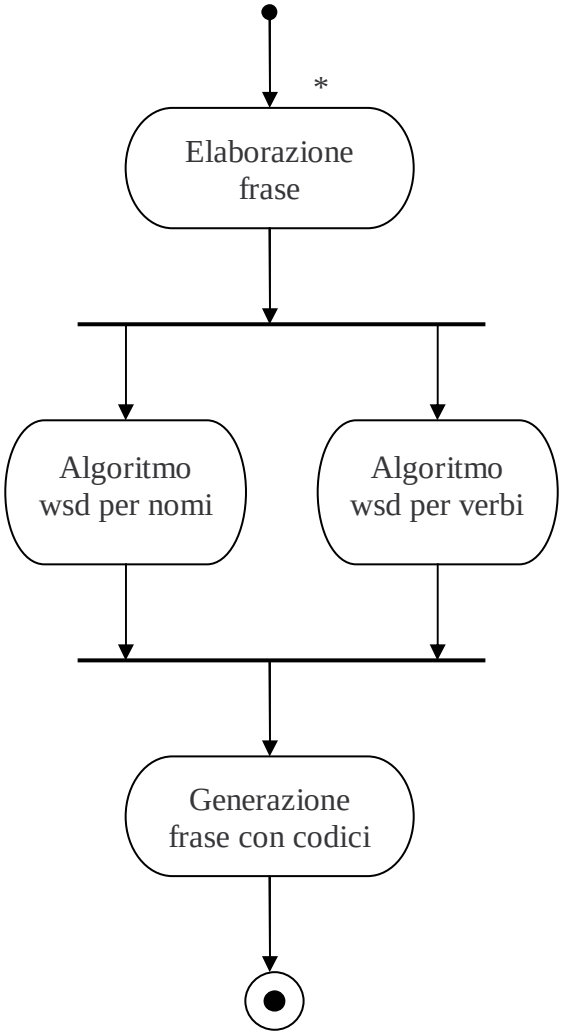


Fig. 3.2 Activity diagram dei passi fondamentali del progetto

Prima di passare a descrivere gli algoritmi utilizzati per il word sense disambiguation, nel prossimo paragrafo verrà introdotto il concetto che sta alla base del calcolo di similarità semantica fra sostantivi: l'iperonimo minimo comune, ossia il concetto meno generale comune a due sostantivi.

3.3 Determinazione del concetto comune a due sostantivi

La determinazione del grado di similarità semantica tra due nomi è basata sulla gerarchia ISA di iperonimi di WordNet.

Percorrendo le gerarchie dei due nomi da confrontare, si verifica se hanno un concetto in comune: in caso affermativo per ognuno dei due nomi si conta il numero di passi necessari per raggiungere il concetto comune. Maggiore è il numero di salti complessivi, minore è la somiglianza di significato tra i due termini perché il concetto comune è più generale. Esempio:

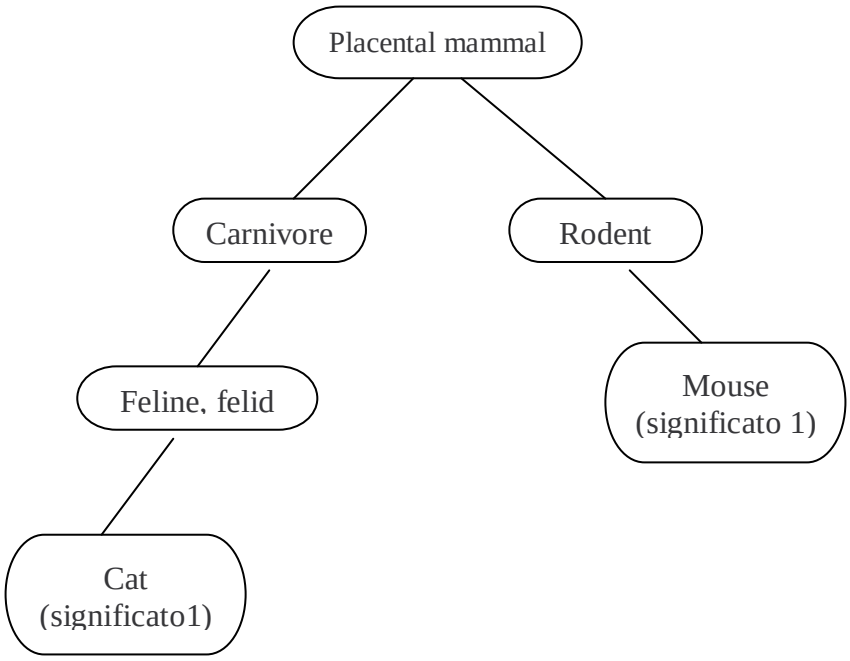


Fig. 3.3 Esempio di concetto comune

Nel caso raffigurato in fig. 3.3, si vede che il concetto comune al significato 1 di *cat* e al significato 1 di *mouse* è rappresentato dal synset che esprime il significato di mammifero placentare; inoltre il numero di passi per raggiungere tale concetto comune è di 3 per *cat* e di 2 per *mouse*.

Il calcolo della similarità semantica di due nomi viene effettuato confrontando tutti i significati di entrambi i nomi e considerando il concetto minimo comune,

cioè quello che rende minimo la lunghezza del cammino tra il primo e il secondo nome all'interno della gerarchia.
Nel progetto non è stato utilizzato come parametro di confronto il semplice numero di salti ma è stata utilizzata la seguente formula logaritmica[20] :

$$sim(c_1, c_2) = - \ln \frac{len(c1, c2)}{2*D}$$

dove c_1 e c_2 sono i due nomi da confrontare, D è la profondità massima raggiungibile dalle gerarchie ISA di WordNet ed è pari a 16, $len(c_1, c_2)$ è la lunghezza del cammino tra c_1 e c_2.
Valutando soltanto il numero di passi che portano da un concetto all'altro non si ottiene un valore di confronto accettabile: a parità di lunghezza minima del cammino non è detto che si abbia un uguale grado di somiglianza tra i termini confrontati. Per questo motivo si normalizza la lunghezza del cammino con il valore 2* D che è la lunghezza massima raggiungibile di un cammino. In questo modo si ottiene un valore di similarità assoluto.

3.4 Algoritmo per il word sense disambiguation dei nomi

Per determinare il significato più appropriato di un nome all'interno di un contesto, è stato sviluppato un algoritmo che si basa su quello presentato nell'articolo di P. Resnik[21] a cui sono state aggiunte delle modifiche.
Il codice dell'algoritmo originale è riportato in figura 3.4, mentre quello dell'algoritmo modificato è in figura 3.5.
Lo sviluppo dell'algoritmo si basa sull'osservazione che quando due parole polisemiche sono simili, il concetto comune dà informazioni su quale significato di ognuna delle parole sia quello più adatto. Data una coppia di nomi possono esserci più coppie di significati che condividono un concetto: meno generico è il concetto comune, maggiore è le probabilità che esso dia informazioni valide sui significati dei nomi presi in considerazione.

Dato un insieme W di nomi, che per noi sono quelli all'interno di una frase ed eventualmente quelli delle frasi adiacenti, e dato l'insieme S dei possibili significati, si vuole determinare una funzione φ che assuma valori compresi tra 0 e 1 e che rappresenti la confidenza con cui si può affermare che il significato $s_{i,j}$ è quello più appropriato per la parola w_i.

L'algoritmo originale prende in considerazione coppie di nomi appartenenti all'insieme W; per ogni coppia considerata si calcola la loro similarità semantica mediante la seguente formula:

$$sim(w_1, w_2) = \max_c [- \ln Pr(c)] , \quad \text{con } c \in \text{subsumers}(w_1, w_2)$$

dove subsumers(w_1, w_2) è l'insieme degli iperonimi di w_1 e di w_2 , mentre Pr(c) è determinata dal seguente rapporto:

$$Pr(c) = \frac{freq(c)}{N} , \qquad \text{con freq(c)} = \sum_{n \in words(c)} count(n)$$

essendo N il numero totale di nomi osservati e words(c) è l'insieme dei nomi che hanno un significato che è iponimo di c.
Pr(c) è dunque una funzione che assume valori crescenti scorrendo la gerarchia degli iperonimi dai concetti più specifici verso quelli più generali. Il segno meno nella formula della similarità porta infine ad avere valori alti nel caso in cui il concetto minimo comune sia specifico e valore bassi quando il concetto è molto generico.

Una volta determinata la similarità semantica, per ogni coppia di parole si determina il concetto minimo comune.
Poi, per ogni significato della prima parola si valuta se il concetto minimo comune è un iperonimo del significato in esame; in caso affermativo si va ad incrementare *support[i,k]* di un valore pari a quello della similarità (support[i,k] è l'elemento

della matrice support relativo alla parola 1 e al significato k). Questo tipo di operazione viene fatto anche per ogni significato della seconda parola.
In questo modo viene dato un "peso" maggiore ai significati che siano iponimi, ossia termini meno generali, del concetto nimino comune.
Ad ogni confronto, vengono incrementati, di un termine pari alla similarità, gli elementi del vettore *normalization* corrispondendenti ai termini della coppia.
Nella seconda parte dell'algoritmo, per ognuno dei significati delle parole di W si calcola la funzione φ di cui si è parlato all'inizio del paragrafo.

Sia W = { w[1], w[2], …. , w[n] } l'insieme dei nomi considerati.

```
for i and j = 1 to n, with i ¡ j
{   v[i, j] = sim(w[i], w[j])
    c[i, j] = concetto minimo comune a w[i] e w[j]
    for k = 1 to num_senses(w[i])
          if c[i,j] è un ipernimo del sense numero k della parola w[i]
             support[i,k] = support[i,k] + v[i,j];
    for q= 1 to num_senses(w[j)
          if c[i,j] è un ipernimo del sense numero q della parola w[j]
             support[j,q] = support[j,q] + v[i,j];
   normalization[i] = normalization[i] + v[i,j];
   normalization[j] = normalization[j] + v[i,j];
}
for i = 1 to n
   for k = 1 to num_senses(w[i])
  { if (normalization[i] != 0.0)
          phi[i, k] = support[i, k] / normalization[i]
     else
          phi[i, k] = 1 / num senses(w[i])
}
```

Fig. 3.4 Algoritmo originale per il word sense disambiguation dei nomi

for i and j = 1 to n, i != j
{ v[i,j] = sim(w[i], w[j]) * G($d_i - d_j$);
  c[i,j] = concetto minimo comune a w[i] e w[j];
  for k = 1 to num_senses(w[i])
    if c[i,j] è un ipernimo del sense numero k della parola w[i]
      support[i,k] = support[i,k] + v[i,j];
  for q= 1 to num_senses(w[j])
    if c[i,j] è un ipernimo del sense numero q della parola w[j]
      support[j,q] = support[j,q] + v[i,j];
  normalization[i] = normalization[i] + v[i,j];
  normalization[j] = normalization[j] + v[i,j];
}
for i = 1 to n
  for k = 1 to num_senses(w[i])
  { if (normalization[i] != 0)

$$phi[i,k] = \alpha * \frac{support[i,k]}{normalization[i]} + \beta * R(z) ;$$

    else

$$phi[i,k] = \alpha * \frac{1}{num\_senses(w[i])} + \beta * R(z) ;$$

  }

Fig. 3.5 Algoritmo modificato per il word sense dismbiguation dei nomi

L'algoritmo modificato mantiene la stessa struttura di quello originale: ciò che lo distingue dal primo è il calcolo della similarità e quello della funzione φ.
Nell'algoritmo modificato, la similarità viene calcolata mediante la formula logaritmica introdotta nel paragrafo 3.3; inoltre, essa viene moltiplicata per un

termine gaussiano che è funzione della differenza delle posizioni dei termini della coppia all'interno della frase:

$$G(x) = 4*N(0,2) + k = 4 * \frac{1}{2*\sqrt{(2*\pi)}} * e^{-\frac{x2}{8}} + k$$

dove $k = 1 - \frac{2}{\sqrt{(2*\pi)}}$, N(0,2) è la distribuzione normale con valor medio uguale a 0 e varianza pari a 2; x è la differenza tra la posizione della parola i e quella della parola j.
In figura 3.6 viene visualizzato l'andamento della gaussiana scelta.

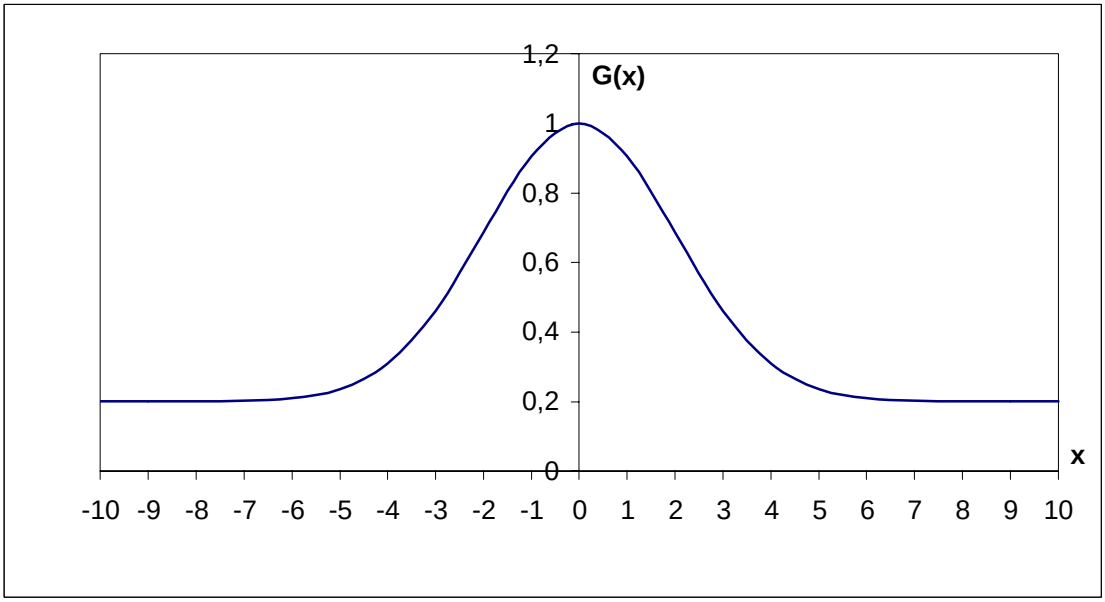


Fig. 3.6 La funzione "gaussiana" usata nell'algoritmo

La misura di similarità φ è stata calcolata come somma di due termini:

$$\varphi(i,k) = \alpha * \frac{support[i,k]}{normalization[i]} + \beta * R(z) , \quad \text{se } normalization[i] \neq 0$$

oppure

$$\varphi(i,k) = \alpha * \frac{1}{num\_senses(w[i])} + \beta * R(z) , \text{ se } normalization[i] = 0$$

il primo tiene conto di quanto la parola considerata è simile dal punto di vista semantico alle altre, mentre il secondo tiene conto della frequenza del significato della parola che si sta esaminando. WordNet infatti numera i significati di una parola in modo crescente a partire da quello più frequente. La funzione R(z), presente nella formula, è una retta con pendenza negativa:

$$R(z) = 1 - 0.8 * \frac{z}{numsenses - 1}$$

z è il numero del significato (per comodità si parte a contare da 0), numsenses è il numero totale dei significati della parola esaminata. Come si può notare questa funzione ha sia dominio che codominio limitati, infatti z assume valori compresi tra 0 e n-1, mentre si è deciso di far assumere a R(z) valori compresi tra 0.2 e 1. Essa è raffigurata in fig 3.8 nel caso in cui numsenses sia pari a 11.

α e β sono due parametri fissati rispettivamente a 0.7 e 0.3 in modo da ottenere un valore di φ compreso tra 0 e 1. Con questo tipo di valori si dà, come ovvio, maggiore importanza al termine che riguarda la somiglianza semantica. Nel caso ci fossero molte parole per cui calcolare la misura di similarità si potrebbe dare ad α un valore maggiore, tenendo conto del fatto che $\alpha + \beta = 1$.

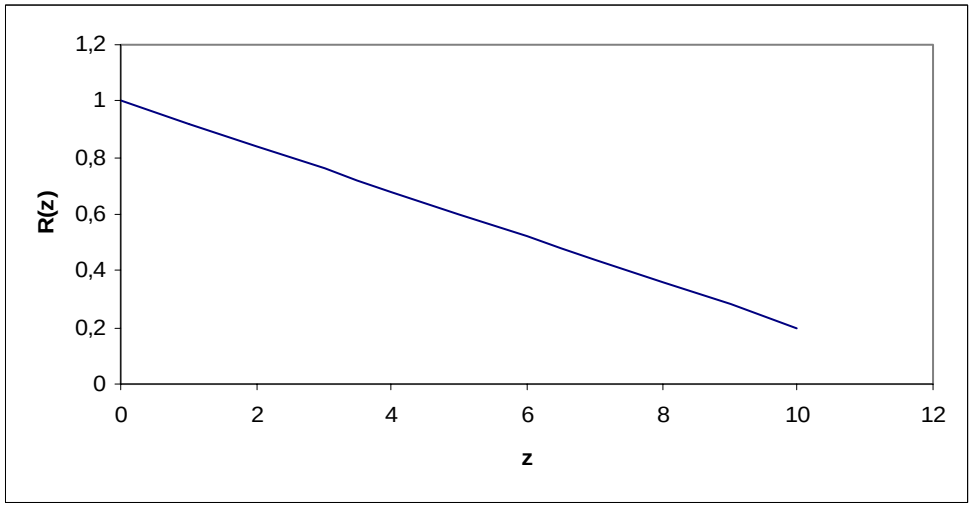


Fig. 3.7 La funzione R(z)

Dopo aver eseguito l'algoritmo, il significato migliore di una parola è quello per cui φ risulta essere maggiore.

Per quanto riguarda l'algoritmo di word sense disambiguation dei nomi, questo può essere calcolato in due modi: valutando solo i nomi della frase esaminata oppure valutando anche i nomi presenti nelle frasi adiacenti.
Nel caso in cui si voglia tener conto anche delle frasi adiacenti, l'algoritmo relativo ai nomi tiene conto anche dei sostantivi in esse inclusi, mentre l'algoritmo relativo ai verbi considera soltanto i verbi che appartengono alla frase originale.
Per quanto riguarda i verbi usare i verbi delle frasi adiacenti per il word sense disambiguation non migliora l'efficacia dell'algoritmo.

3.5 Algoritmo per il word sense disambiguation dei verbi

I verbi giocano un ruolo importante nel linguaggio umano; essi vincolano e mettono in relazione le varie parti di una frase: ad esempio legano il soggetto con il complemento oggetto.
Alcuni verbi assumono uno dei loro significati in base alla struttura grammaticale che viene usata nella frase in cui sono inseriti; altri verbi assumono un loro particolare significato solo se sono inseriti in uno specifico contesto.

Nell'algoritmo utilizzato per il word sense disambiguation delle forme verbali è stato usato un algoritmo che va applicato non solo ai sostantivi della frase ma anche a quelli compresi negli esempi d'uso dei verbi.
WordNet, infatti, nella definizione che fornisce di ogni significato include anche qualche frase di esempio che possa chiarire ulteriormente la spiegazione data.
In figura 3.8 viene riportata la definizione del significato #2 del verbo *look*. Come si può notare la definizione è divisa in due parti: la prima (in italico nella figura) è una breve descrizione del significato, la seconda contiene gli esempi d'uso.

Sense 2

look, appear, seem -- (*give a certain impression or have a certain outward aspect;*

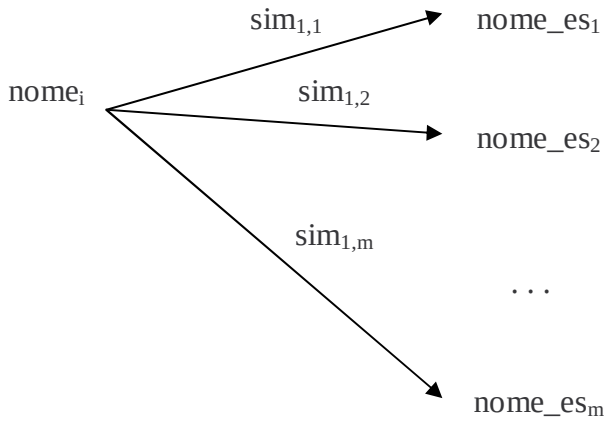
"She seems to be sleeping"; "This appears to be a very difficult problem"; "This project looks fishy"; "They appeared like people who had not eaten or slept for a long time")

Fig. 3.8 Esempio di definizione di WordNet

Vediamo ora di illustrare l'algoritmo di word sense disambiguation adottato per i verbi.

Per ogni significato di un verbo, si estraggono i nomi presenti nella definizione di tale significato, sia quelli della descrizione che quelli presenti nelle frasi di esempio. Sia *nomi_es*(v,k) l'insieme dei nomi della definizione del significato k del verbo v; sia N l'insieme dei nomi presenti nella frase in cui compare il verbo v. Usando l'esempio di figura 3.8, si ha *nomi_es*(look, 2) = { impression, aspect, problem, project, people, time }.

Ogni nome appartenente a N viene confrontato con tutti i nomi di nomi_es(v,k), come illustrato in figura 3.9:



i = 1, 2, ..., n
con n = numero di nomi presenti nella frase che contiene il verbo;
$nome\_es_j$ = nome appartenente a nomi_es(v,k).

Fig. 3.9 Il confronto tra i nomi della frase e quelli delle frasi di esempio

Dalla figura precedente si nota che ogni confronto produce il calcolo della similarità tra i due termini. Tale similarità è la stessa usata nel caso del word sense disambiguation dei nomi (vedi paragrafo 3.3), ma in questo caso viene normalizzata dal valore più alto che essa può assumere, ossia quello che si ottiene in corrispondenza del numero minimo di salti che si possono compiere all'interno della gerarchia ISA. Questo numero è pari a 2, infatti per ogni synset bisogna compiere almeno un salto per giungere al concetto comune.

$$\text{sim}(\text{nome}_i , \text{nome\_es}_j) = \frac{-\ln(\text{numero salti}/(2*D))}{-\ln(1/D)}$$

Nel caso i due nomi non avessero un iperonimo in comune si pone la similarità pari a 0.
Dopo aver calcolato la similarità per ogni coppia in questione, si determina per ogni nome appartenente il seguente valore:

$$\text{max}_i = \max\{ \text{sim}(\text{nome}_i , \text{nome\_es}_j), \text{con } j = 1,2, ..., m\}$$

dove i = 1, 2, ..., n ; j = 1,2, ..., m ; n è la dimensione dell'insieme N,m è quella dell'insieme nomi_es(v,k).

Analogamente al caso riguardante i nomi, ora siamo in grado di fornire una funzione φ(v,k) che rappresenti la confidenza con cui si può dire quanto il significato k sia appropriato nella frase in cui è inserito. Questa funzione viene calcolata mediante la seguente formula:

$$\varphi(v, k) = R(k) * \sum_i \frac{G(d_i) * \max_i}{\sum_i G(d_i)}$$

dove i = 1, 2, ..., n.

G è la stessa funzione di tipo gaussiano utilizzata nell'algoritmo di word sense disambiguation dei nomi; essa viene calcolata in d_i che è la differenza fra la posizione del nome $nome_i$ e quella del verbo v all'interno della frase.
Il termine a denominatore è la somma delle gaussiane calcolate per ogni nome appartenente a N; esso serve a produrre un valore di φ compreso tra 0 e 1.
R(k) è una funzione analoga alla funzione R(z) presentata nel paragrafo relativo all'algoritmo dei nomi.
Essa è infatti una retta con una pendenza maggiore rispetto alla precedente e viene formulata nel seguente modo:

$$R(k) = 1 - 0.9 * \frac{k}{numsenses - 1}$$

k è il numero del significato di v che si sta valutando, numsenses è il numero totale di significati del verbo v.
Anche in questo caso il dominio e il codominio sono limitati: il dominio varia tra 0 e il numero di significati del verbo v meno 1, il codominio varia tra 0.1 e 1.
In figura 3.10 è visualizzata la retta nel caso in cui numsenses sia pari a 11.

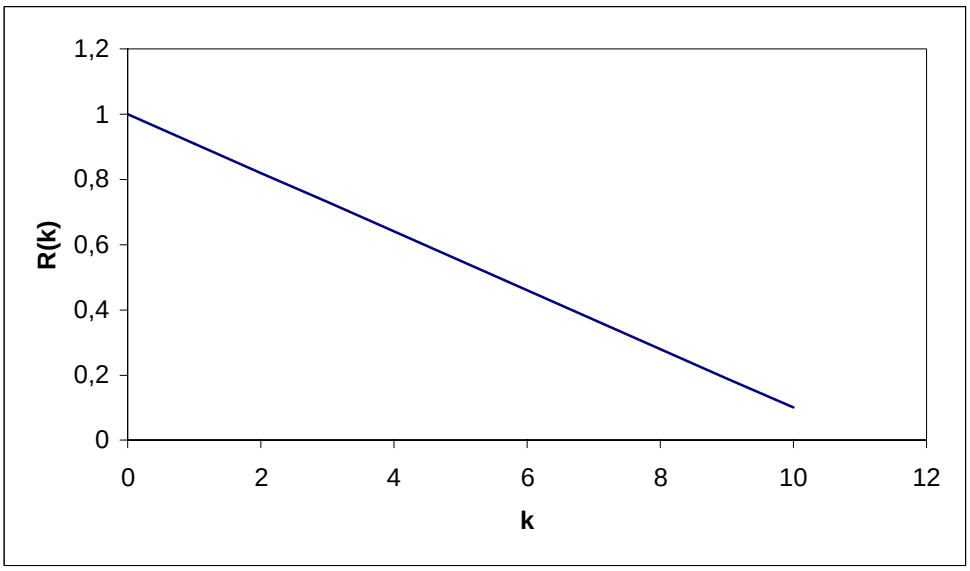


Fig. 3.10 La funzione R(k)

Poichè esistono in WordNet verbi con oltre 30 significati, si è deciso di introdurre una condizione che rendesse più efficiente l'algoritmo. Visto che la retta R(k) è una funzione di tipo decrescente si ha:

$$R(k) > R(k + 1) > R(k + 2) > \ldots$$

Se risulta vera la condizione:

$$\varphi(v, k) > R(k + 1)$$

allora si ha che:

$$\varphi(v, k) = R(k)*T(v, k) > R(k+1) > R(k+1)*T(v, k+1) = \varphi(v, k+1)$$

Si può perciò smettere di calcolare φ(v,k) per k > k+1, infatti per tale intervallo di valori la funzione φ assumerà sempre valori minori rispetto a quello calcolato per il significato k. É perciò inutile calcolare il valore di φ per tali significati visto che non verrebbero comunque scelti come significati migliori.

Come nell'algoritmo riguardante i nomi, una volta calcolato φ, viene scelto come significato migliore del verbo v quello che risulta avere un valore di φ maggiore degli altri.

3.6 Generazione della frase con i codici di WordNet

Una volta eseguiti i due algoritmi si possono riunire i risultati ottenuti.
Ad ogni parola inclusa nella frase corrisponde il numero del significato che è risultato essere più appropriato; con queste informazioni è facile ricavare il codice univoco assegnato da WordNet ai synset e sostituire tali codici alle corrispondenti parole della frase.

Capitolo 4

Il progetto del software

Nel precedente capitolo è stato illustrato il procedimento scelto per la risoluzione di ambiguità semantiche dei termini contenuti in una frase.
In questo capitolo, verrà descritto come è stato implementato il progetto dal punto di vista software.

4.1 I package del progetto EXTRA

Il programma scritto per questa tesi usando il linguaggio Java (vedi appendice A), implementa un modulo che realizza il word sense disambiguation; tale modulo fa parte del progetto EXTRA che ha come scopo la ricerca di similarità tra frasi. Viene perciò aggiunta ad EXTRA una funzionalità semantica.
In figura 4.1 è mostrato il package diagram del progetto EXTRA, dove il package realizzato in questa tesi è evidenziato in grigio.

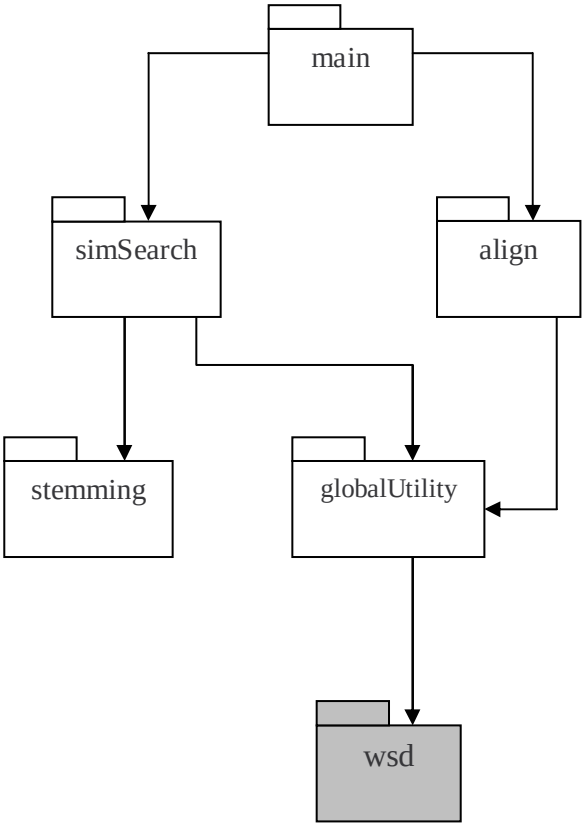


Fig. 4.1 Package diagram del progetto EXTRA

Il progetto EXTRA è organizzato originaramente in cinque package:

- *simSearch*, contenente le classi che realizzano la ricerca di similarità e gestiscono la Translation Memory;
- *align*, che realizza l'allineamento delle frasi e delle parole
- *stemming*, che effettua lo stemming delle frasi;
- *globalUtility*, contenente utilità comuni sfruttate dai vari package;
- *main*, il package principale, che utilizza le funzionalità degli altri e fornisce la comune interfaccia grafica.

A questi package va aggiunto il modulo realizzato per questa tesi: è costituito da un unico package *wsd* che contiene le seguenti classi:

- *Start*: è la classe principale che richiama le altre;
- *Disambiguation*: implementa attraverso i suoi metodi gli algoritmi di word sense disambiguation;
- *Elaborazioni*: qui le frasi vengono trasformate in modo da poter essere poi usate per risolvere gli algoritmi di risoluzione di ambiguità semantica;
- *Risultati*: questa classe serve a riportare il risultato della ricerca di un iperonimo comune a due concetti;
- *Liste*: serve a elencare le liste di nomi e verbi contenuti in una frase e le relative posizioni all'interno della frase.

4.2 Le classi del progetto

4.2.1 Classe Disambiguation

La classe *Disambiguation* è quella che implementa gli algoritmi per la risoluzione di ambiguità semantiche delle parole.
Gli attributi della classe sono:

- D: è il numero massino di livelli raggiungibili dalle gerarchie di WordNet;
- PI: è la costante $\pi = 3.14$;
- lista: contiene la lista degli iperonimi del significato che si sta analizzando;

In figura 4.2 è indicata la struttura della classe Disambiguation.

I metodi della classe sono:

- ancestor(PointerTarget, PointerType): riporta la lista di tutti gli iperonimi del significato passato come primo parametro;
- codiciSenses(String, Vector, Vector): è il metodo che dopo aver risolto le ambiguità dei nomi e dei verbi sostituisce ad ogni nome della frase "parserizzata" e "stemmizzata" i rispettivi codici WordNet dei significati che sono risultati migliori nell'analisi eseguita;
- esempiUuso(String, int): dato un verbo e il numero di un suo significato ne estrae le relative frasi di esempio;

- estraiNomi(String, Vector): data una frase “parserizzata” e “stemmizzata” estrae la lista dei nomi in essa contenuti;
- nomiUuso(String, int, JMontyTagger): dato un verbo e il numero di un suo significato estrae i nomi contenuti nelle relative frasi di esempio;
- pesoVerbi(String, int, int, Vector, int, DictionaryDatabase, JMontyTagger): calcola il peso φ di un significato di un verbo (vedi paragrafo 3.5);
- wsdNomi(Vector, DictionaryDatabase): è il metodo che implementa l’algoritmo di word sense disambiguation dei nomi;
- wsdVerbi(Vector, Vector, JMontyTagger): è il metodo che implementa l’algoritmo di word sense disambiguation dei nomi.

Disambiguation
int D; double PI; Vector lista;
Vector **ancestor**(PointerTarget, PointerType); String **codiciSenses**(String, Vector, Vector); Vector **esempiUso**(String, int); List **estraiNomi**(String, Vector); Vector **nomiUso**(String, int, JMontyTagger); double **pesoVerbi**(String, int, int, Vector, int, DictionaryDatabase, JMontyTagger); Vector **wsdNomi**(Vector, DictionaryDatabase); Vector **wsdVerbi**(Vector, Vector, JMontyTagger);

Fig. 4.2 Struttura della classe *disambiguation*

4.2.2 Classe Elaborazioni

La classe *Elaborazioni* contiene vari metodi per trasformare la frase orignale in modo da poter essere poi utilizzata dalla classe wsd che implementa gli algoritmi di word sense disambiguation.
Gli attributi della classe sono:

- gerarchia: è un vettore che contiene la gerarchia degli iperonimi di una parola (la prima di una coppia di termini da confrontare); ogni elemento del vettore contiene un iperonimo e il livello a cui si trova all'interno della gerarchia;
- gerarchia2: è il vettore che contiene la gerarchia degli iperonimi della seconda delle parole da confrontare; è strutturata come il precedente attributo;
- indice: serve durante la costruzione della gerachia ad indicare il livello di ricorsione a cui si è giunti;
- livello: indica durante la costruzione della gerarchia il livello a cui si è arrivati.

In figura 4.3 è rappresentata la struttura della classe Elaborazioni.

I metodi della classe sono:

- cercaLivello(Vector, int): questo metodo estrae "posizione" da una stringa del tipo "parola/posizione" contenuta nel vettore passato come primo parametro alla posizione data dal secondo parametro; la stringa "posizione" viene poi convertita in un intero;
- cercaParola(Vector, int): questo metodo estrae "parola" da una stringa del tipo "parola/posizione" contenuta nel vettore passato come primo parametro alla posizione data dal secondo parametro;
- cercaParola(Vector, String): verifica se la stringa è già stata inserita nel vettore;

- cercaPosiz(String, String, int): cerca la posizione di una parola (primo argomento) all'interno di una frase (secondo argomento) a partire dalla posizione data dal terzo parametro;
- concettoComune(Vector, Vector): cerca, se esiste, un iperonimo comune alle due gerarchie memorizzate nei vettori passati come parametri;
- confrontaStringhe(String, String, DictionaryDatabase): cerca, se esiste, l'iperonimo minimo comune alle parole passate come primi due parametri;
- contaSalti(Vector, String): conta il numero di salti all'interno della gerarchia, memorizzata nel vettore, che servono ad arrivare al concetto descritto dalla stringa passata come secondo parametro;
- elabora(String, String, int): in base ai tag della frase parserizzata (primo parametro), estrae le liste dei nomi e dei verbi contenuti nella frase originale (secondo parametro) e la frase "parserizzata" e stemmizzata";il terzo parametro indica il numero della frase da analizzare;
- elaboraFrasePars(String, String): come il metodo elabora ma estrae solo le liste dei nomi e dei verbi;
- hypernyms(PointerTarget, PointerType, String): estrae la gerarchia degli iperonimi del significato passato come primo parametro;
- ordina(Vector): il vettore passato come parametro contiene stringhe del tipo "parola/posizione"; gli elementi del vettore vengono ordinati in modo crescente su posizione;
- parsing(String, JMontyTagger): effettua il parsing della frase passata come primo parametro;
- parsing(Vector, JMontyTagger): effettua il parsing delle frasi contenute nel vettore passato come primo parametro;
- stemming(Vector): esegue lo stemming, mediante il modulo implementato da F. Gavioli, delle frasi contenute nel vettore passato come parametro.

Elaborazioni
Vector gerarchia; Vector gerarchia2; int indice; int livello;
int **cercaLivello**(Vector, int); String **cercaParola**(Vector, int); boolean **cercaParola**(Vector, String); int **cercaPosiz**(String, String, int); String **concettoComune**(Vector, Vector); Vector **confrontaStringhe**(String, String, DictionaryDatabase); int **contaSalti**(Vector, String); liste **elabora**(String, String, int); liste **elaboraFrasePars**(String, String); void **hypernyms**(PointerTarget, PointerType, String); void **ordina**(Vector); String **parsing**(String, JMontyTagger); Vector **parsing**(Vector, JMontyTagger); Vector **stemming**(Vector);

Fig. 4.3 Struttura della classe E*laborazioni*

4.2.3 Altre classi

Le restanti classi non sono importanti dal punto di vista dell'implementazione del progetto, perciò verrà riportata una breve descrizione insieme agli schemi delle loro strutture.

La classe *Risultati* serve a riportare i risultati del confronto fra gerarchie di due concetti; essa è strutturata come in figura 4.4.
Come la precedente classe, anche la classe *Liste* serve a riportare un tipo di risultato: le liste dei nomi e dei verbi; essa è strutturata come in figura 4.5.

Risultati
String ipernimo; int numsalti;

risultati(String, int); // è il costruttore
String **getIpernimo**();
int **getSalti**();

Fig. 4.4 Struttura della classe *Risultati*

Liste
Vector listaNomi; Vector listaVerbi; String frase;

liste(Vector nomi, Vector verbi);
liste(String frase1, Vector nomi, Vector verbi);
Vector getListaNomi();
Vector getListaVerbi();
String getFrase();

Fig. 4.5 Struttura della classe *Liste*

Infine la classe Start è la classe principale da cui vengono richiamate le altre classi.

4.3 Il flusso dei dati

Per rendere più chiaro il funzionamento del programma si riportano i data flow diagrams delle operazioni svolte. Per quanto riguarda i metodi, viene riportato oltre al nome, la classe di appartenenza. Per questioni di spazio la classe Elaborazioni verrà indicata con ‘Elab’, mentre la classe Disambiguation verrà indicata con ‘Disamb’.

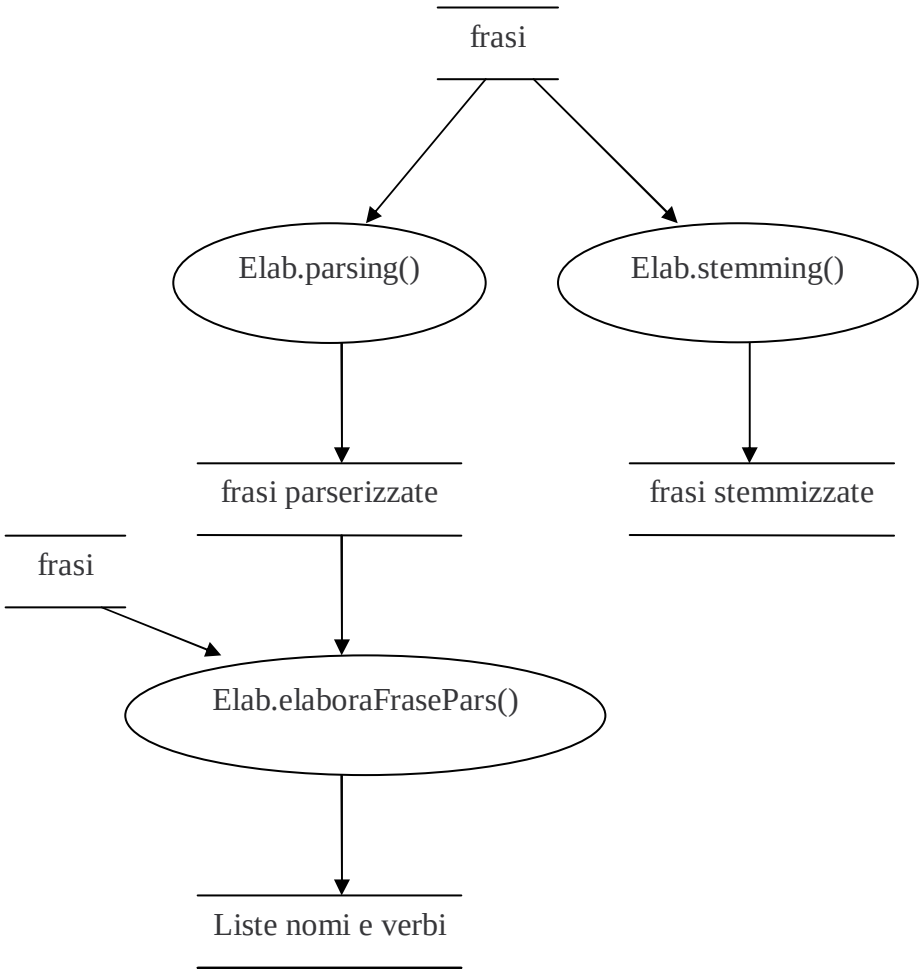


Fig. 4.6 Data flow diagram delle prima fase di trasformazioni

In figura 4.6 è illustrata la prima fase di trasformazioni subite dalle frasi in ingresso. Innanzitutto le frasi contenute nel documento da analizzare vengono “parserizzate” e “stemmizzate”. Mediante l’operazione di parsing è possibile estrarre dalle frasi le liste dei nomi e dei verbi in esse contenute.

A questo punto la frase è pronta per essere “disambiguata”. In figura 4.7 viene visualizzato il flusso di dati che interessa la parte del programma che implementa gli algoritmi di risoluzione di ambiguità semantiche.
Nel diagramma i termini ris_nomi e ris_verbi sono due vettori che contengono i risultati degli algoritmi di word sense disambiguation rispettivamente dei nomi e dei verbi; ogni elemento di questi vettori contiene una parola (nome o verbo) e il codice WordNet del significato che è risultato essere più appropriato per tale parola. Alla fine di tutto il procedimento si ottiene la frase a cui ad ogni parola normalizzata viene sostituito il corrispondente codice.

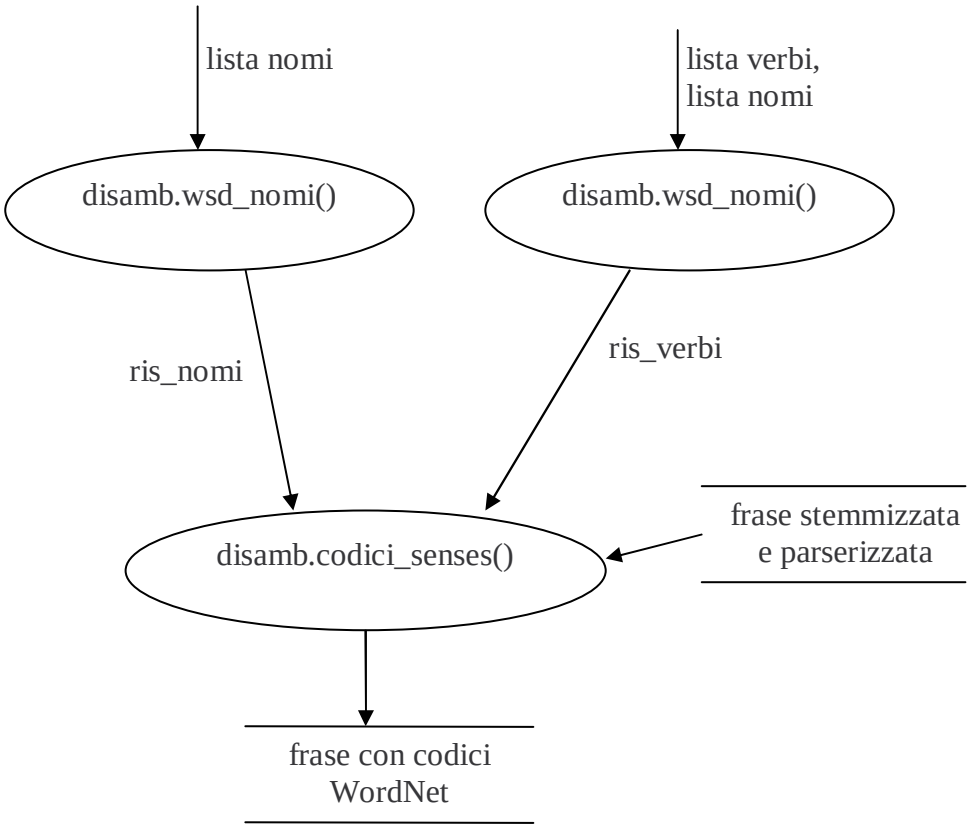


Fig. 4.7 Data flow diagram della risoluzione di ambiguità semantiche

Capitolo 5

Prove sperimentali e conclusioni

In questo capitolo vengono mostrati i risultati ottenuti utilizzando il procedimento scelto per il word sense disambiguation di termini in lingua inglese.
In particolare verrà analizzata l'efficacia delle prestazioni nel caso in cui gli algoritmi vengano applicati ad una frase alla volta e nel caso in cui si applichino anche alle frasi adiacenti.
I test sono stati eseguiti su un computer con processore AMD Athlon 1500+ (1,33 GHz) con 256 MB di memoria e hard disk da 20 GB.

5.1 Collezioni usate nei test

Per eseguire le prove di efficacia sono stati utilizzati due diversi tipi di documenti: uno di tipo tecnico e uno di tipo letterario. Per il primo tipo di documenti sono state usate due collezioni di dati:

- collezione "DPaint": comprende frasi tratte da un manuale tecnico di un prodotto software;
- collezione "elettrodomestici": contiene frasi tratte da manuali di istruzioni di elettrodomestici.

Per quanto riguarda il secondo tipo di documenti, è stata usata una collezione di dati:

- collezione "opere letterarie": contiene frasi tratte da celebri opere letterarie.

Questo tipo di distinzione è stato fatto per testare l'efficacia degli algoritmi in ambiti diversi; poichè WordNet possiede un dizionario non specializzato, si è voluto verificare le differenze di prestazioni tra frasi di tipo "generico" come quelle presenti nelle opere letterarie e frasi contenenti termini specifici di un determinato settore.
Nel caso dei verbi è più difficile determinare quale sia il significato più giusto infatti WordNet in molti casi, assegna allo stesso verbo significati simili. Per questo motivo l'efficacia dell'algoritmo dei verbi è stata valutata a parte con una serie di esempi creati ad hoc.

5.2 Efficacia dell'algoritmo dei nomi

La risoluzione di ambiguità semantiche di sostantivi è stata testata sulle tre collezioni di dati descritte nel paragrafo precedente. Per ognuna delle collezioni si è preso in considerazione un numero di frasi pari a 30. L'efficacia è stata valutata in tre diversi casi, ognuno per un valore diverso del parametro *primaedopo* (indica quante frasi adiacenti valutare insieme a quella sorgente):

a) *primaedopo* = 0: in questo caso le frasi vengono valutate una alla volta, cioè non si considerano le frasi adiacenti;
b) *primaedopo* = 1: vengono considerate la frase precedente e quella successiva, oltre a quella in ingresso;
c) *primaedopo* = 2: vengono considerate le due frasi precedenti e le due successive, oltre a quella sorgente.

In tutti i casi si è valutata la media delle percentuali di successo.

Caso a)
Come si può vedere dal grafico di figura 5.1 nei casi relativi alle due collezioni di tipo tecnico si raggiunge un'efficacia del 70% nel caso della collezione "DPaint" e del 68,6 % nel caso relativo al manuale di istruzioni dell'elettrodomestico. La collezione relativa alle opere letterarie ha un'efficacia leggermente migliore, pari al 79,5%. Questo risultato è giustificato dal fatto che nelle collezioni tecniche

sono presenti termini specifici dell'ambiente tecnico-informatico che non sempre sono memorizzati nel database di WordNet.

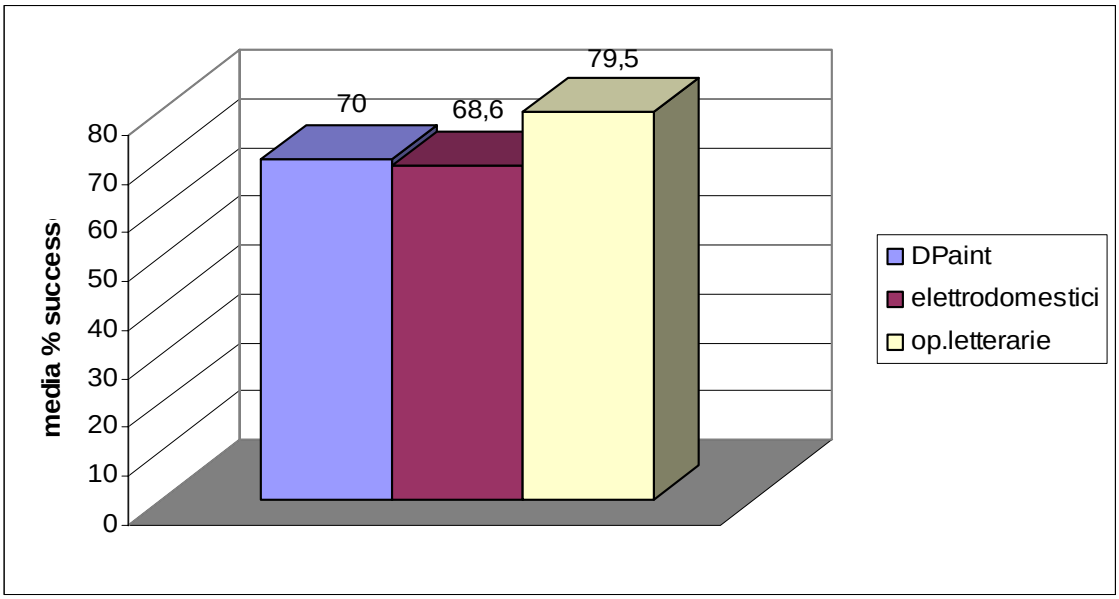


Fig. 5.1 Media delle percentuali di successo (caso *primaedopo* = 0)

Casi b) e c):

Nei casi in cui il parametro *primaedopo* assuma i valori 1 e 2, si ottengono risultati sostanzialmente analoghi in quanto a prestazioni di efficacia. In entrambi i casi infatti si ha un miglioramento dei risultati.

Per quanto riguarda le collezioni di tipo tecnico si ha un incremento delle prestazioni che va dal 3,4% nel caso della collezione relativa al manuale di istruzioni per l'elettrodomestico al 9% circa della collezione "DPaint"; nel caso delle opere letterarie l'incremento è pari a circa l'11%.

In figura 5.2 sono messi a confronto la media delle percentuali di successo del caso a) con quelle dei casi b) e c).

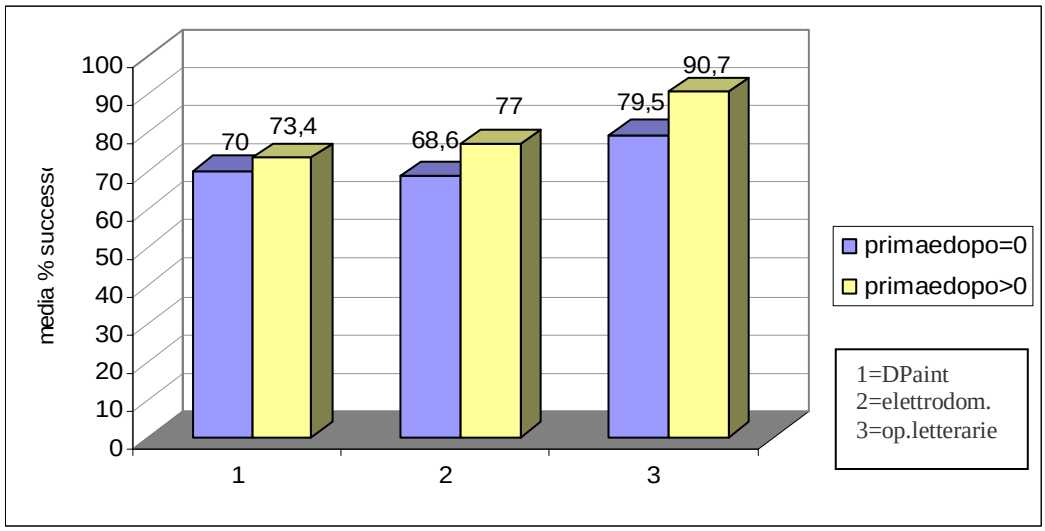


Fig. 5.2 Media delle percentuali di successo a confronto

Vengono ora riportati alcuni esempi significativi.

a) Consideriamo le seguenti frasi prese dalla collezione “elettrodomestici”:

```
A timer can be used to any selected burner.
Before any maintenance on the appliance disconnect it from the
electrical power supply.
Do not use abrasive products bleach oven cleaner spray or pan
scourers.
```

Nel caso parametro *primaedopo* = 0, si ottengono i seguenti risultati:

- nella prima frase:
  - *timer*, significato scelto: `a timepiece that measures a time interval and signals its end`, con un valore di φ pari a 1,0;
  - *burner*, significato scelto: `an apparatus for burning fuel (or refuse)`, con un valore di φ pari a 1,0;
- nella seconda frase:
  - *maintenance*, significato scelto: `the act of sustaining`, con un valore di φ pari a 0,76;

- *appliance*, significato scelto: `a device that very useful for a particular job`, con un valore di φ pari a 1,0;
- *power*, significato scelto: `possession of controlling influence`, con un valore di φ pari a 0,61;
- *supply*, significato scelto: `an amount of something available for use`, con un valore di φ pari a 0,60;

- nella terza frase:
  - *product*, significato scelto: `commodities offered for sale`, con un valore di φ pari a 0,87;
  - *bleach*, significato scelto: `an agent that makes things white or colorless`, con un valore di φ pari a 0,88;
  - *oven*, significato scelto: `kitchen appliance used for baking or roasting`, con un valore di φ pari a 1,0;
  - *spray*, significato scelto: `a dispenser that turns a liquid (such as perfume) into a fine spray`, con un valore di φ pari a 0,83;
  - *pan*, significato scelto: `shallow container made of metal`, con un valore di φ pari a 0,84;
  - *scourer*, significato scelto: `a place that is scoured (especially by running water)`, con un valore di φ pari a 1,0.

Nella prima frase viene assegnato correttamente solo il significato del termine *timer*, nella seconda frase vengono assegnati correttamente i significati dei termini *appliance* e *supply*, nella terza frase vengono assegnati in modo appropriato tutti i significati dei termini tranne quello di *scourer*.

Nel caso in cui il parametro primaedopo assuma il valore 1, si ottengono i seguenti risultati:

- nella seconda frase, viene assegnato correttamente anche il significato di maintenance, che risulta essere: `activity involved in maintaining something in good working order;` il valore di φ è pari a 0,59;

- i risultati relativi agli altri termini non variano.

b) Consideriamo ora un blocco di frasi prese dalla collezione DPaint:

```
After spending a little time with this manual you will be able to
use the program to create colorful graphics in a fraction of the
time it would take using more traditional techniques.
We have organized the information so you can quicly learn what you
need to use the program in a manner best suited to your style and
experience.
And this lets you build up a library of images or clip art to use
in future designs.
```

Nel caso parametro *primaedopo* = 0, si ottengono i seguenti risultati:

- nella prima frase:
  - *time*, significato scelto: `a suitable moment,` con un valore di φ pari a 0,49;
  - *program*, significato scelto: `a radio or television show,` con un valore di φ pari a 0,74;
  - *graphic*, significato scelto: `photographs or other visual representations in a printed publication,` con un valore di φ pari a 0,76;
  - *fraction*,, significato scelto: `the quotient of two rational numbers,` con un valore di φ pari a 0,49;
  - *time*, significato scelto: `a suitable moment,` con un valore di φ pari a 0,52;
  - *tecnique*, significato scelto: `a practical method or art applied to some particular task,` con un valore di φ pari a 1,0;
- nella seconda frase:
  - *information*, significato scelto: `a message received and understood that reduces the recipient's uncertainty,` con un valore di φ pari a 0,72;

- *program*, significato scelto: `a course of academic studies`, con un valore di φ pari a 0,66;
- *man*, significato scelto: `all of the inhabitants of the earth`, con un valore di φ pari a 0,61;
- *style*, significato scelto: `a particular kind (as to appearance)`, con un valore di φ pari a 0,57;
- *experience*, significato scelto: `the accumulation of knowledge or skill that results from direct participation in events or activities`, con un valore di φ pari a 1;

- nella terza frase:
  - *library*, significato scelto: `a room where books are kept`, con un valore di φ pari a 0,92;
  - *image*, significato scelto: `a visual representation of an object or scene or person produced on a surface`, con un valore di φ pari a 0,68;
  - *clip*, significato scelto: `a metal frame or container holding cartridges; can be inserted into an automatic gun`, con un valore di φ pari a 0,92;
  - *art*, significato scelto: `the products of human creativity; works of art collectively`, con un valore di φ pari a 0,91;
  - *future*, significato scelto: `bulk commodities bought or sold at an agreed price for delivery at a specified future date`, con un valore di φ pari a 0,69;
  - *design*, significato scelto: `a decorative or artistic work`, con un valore di φ pari a 0,69;

Nel caso in cui il parametro primaedopo assuma un valore diverso da zero, non si ottengono miglioramenti.

c) Prendiamo in condìsiderazione le seguenti frasi tratte dalla collezione “opere letterarie”:

Once there was a gentleman who married for his second wife the proudest and most haughty woman that was ever seen.
She employed her in meanest work of the house she scoured the dishes tables etc and scrubbed madam's chamber and those of misses her daughters she lay up in a sorry garret upon a wretched straw bed while her sisters lay in fine rooms with floors all inlaid upon beds of the very newest fashion and where they had looking-glasses so large that they might see themselves at their full length from head to foot.
When she had done her work she used to go into the chimney-corner and sit down among cinders and ashes which made her commonly be called a cinder maid.

Nel caso parametro *primaedopo* = 0, si ottengono i seguenti risultati:

- nella prima frase:
  - *gentleman*, significato scelto: a man of refinement, con un valore di φ pari a 1.0;
  - *wife*, significato scelto: a married woman; a man's partner in marriage, con un valore di φ pari a 1,0;
  - *woman*, significato scelto: an adult female person (as opposed to a man), con un valore di φ pari a 1,0;
- nella seconda frase:
  - *work*, significato scelto: a product produced or accomplished through the effort or activity or agency of a person or thing, con un valore di φ pari a 0,65;
  - *house*, significato scelto: a dwelling that serves as living quarters for one or more families, con un valore di φ pari a 0,57;
  - *dish*, significato scelto: a piece of dishware normally used as a container for holding or serving food, con un valore di φ pari a 0,52;
  - *table*, significato scelto: a piece of furniture having a smooth flat top supported by one or more vertical legs, con un valore di φ pari a 0,53;

- *madam*, significato scelto: `a woman of refinement`, con un valore di φ pari a 1,0;
- *chamber*, significato scelto: `a room where a judge transacts business`, con un valore di φ pari a 0,75;
- *daughter*, significato scelto: `a female human offspring`, con un valore di φ pari a 1,0;
- *garret*, significato scelto: `floor consisting of open space at the top of a house just below roof; often used for storage`, con un valore di φ pari a 1,0;
- *straw*, significato scelto: `plant fiber used e.g. for making baskets and hats or as fodder`, con un valore di φ pari a 0,66;
- *bed*, significato scelto: `a piece of furniture that provides a place to sleep`, con un valore di φ pari a 0,63;
- *sister*, significato scelto: `a female person who has the same parents as another person`, con un valore di φ pari a 0,82;
- *room*, significato scelto: `an area within a building enclosed by walls and floor and ceiling`, con un valore di φ pari a 0,77;
- *floor*, significato scelto: `structure consisting of a room or set of rooms comprising a single level of a multilevel building`, con un valore di φ pari a 0,67;
- *bed*, significato scelto: `a piece of furniture that provides a place to sleep`, con un valore di φ pari a 0,63;
- *fashion*, significato scelto: `a manner of performance`, con un valore di φ pari a 0,81;
- *glass*, significato scelto: `a glass container for holding liquids while drinking`, con un valore di φ pari a 0,72;
- *length*, significato scelto: `the linear extent in space from one end to the other; the longest horizontal dimension of something that is fixed in place`, con un valore di φ pari a 0,54;
- *head*, significato scelto: `a person who is in charge`, con un valore di φ pari a 0,53;

- nella terza frase:
  - *work*, significato scelto: `a product produced or accomplished through the effort or activity or agency of a person or thing,` con un valore di φ pari a 0,78;
  - *chimney*, significato scelto: `a vertical flue that provides a path through which smoke from a fire is carried away through the wall or roof of a building,` con un valore di φ pari a 1,0;
  - *corner*, significato scelto: `a projecting part that is corner-shaped,` con un valore di φ pari a 0,67;
  - *cinder*, significato scelto: `a fragment of incombustible matter left after a wood or coal or charcoal fire,` con un valore di φ pari a 1,0;
  - *ash*, significato scelto: `the residue that remains when something is burned,` con un valore di φ pari a 1,0;
  - *cinder*, significato scelto: `a fragment of incombustible matter left after a wood or coal or charcoal fire,` con un valore di φ pari a 1,0;
  - *maid*, significato scelto: `a female domestic,` con un valore di φ pari a 1,0;

In questo caso nella prima frase vengono assegnati in modo corretto i significati di tutti i termini. Nella seconda frase l'assegnamento è giusto per i termini *work*, *madam*, *daughter*, *garret*, *straw*, *bed*, *sister*, *room*, *floor*, *fashion*, *lenght*; quando si applica il valore 1 al parametro *primaedopo* il numero degli assegnamenti corretti aumenta, infatti vengono correttamente identificati i significati di tutte le parole tranne chamber e head. Nella terza frase, l'unica parola il cui significato non viene assegnato nel modo giusto è corner; anche aumentando il valore di *primaedopo* non si ottengono miglioramenti.

5.3Efficacia dell'algoritmo dei verbi

Valutare l'efficacia dell'algoritmo relativo ai verbi non è semplice soprattutto quando si ha a che fare con verbi che possono assumere molti significati di cui buona parte simili tra loro; anche quando si considerano verbi come "be" o "have" non è immediato riconoscere quale sia il significato più adatto.
In questo paragrafo si presentano perciò alcuni esempi significativi che possano mostrare l'efficacia dell'algoritmo.

- Frase:
The great musician Beethoven composed nine symphonies and many other musical composition.

Tra i 6 significati del verbo "compose" assegnati da WordNet (vedi figura 5.3), l'algoritmo sceglie giustamente il secondo con valore di φ pari a 0,80.

The verb compose has 6 senses (first 5 from tagged texts):
1. (14) compose -- (form the substance of; "Greed and ambition composed his personality")
2. (5) compose, write -- (write music; "Beethoven composed nine symphonies")
3. (4) write, compose, pen, indite -- (produce a literary work; "She composed a poem"; "He wrote four novels")
4. (3) compose, compile -- (put together out of existing material; "compile a list")
5. (1) compose -- (calm (someone, esp. oneself); make quiet; "She had to compose herself before she could reply to this terrible insult")
6. frame, outline, compose, draw up -- (draw up the plans or basic details for; "frame a policy")

Fig.5.3 I significati del verbo *compose*

- Frase:
Once there was a gentleman who married for his second wife the proudest and most haughty woman that was ever seen.

In questo caso nella frase sono presenti più verbi; l'algoritmo dà i seguenti risultati:

-> verbo be, significato scelto: `be identical to; be someone or something` con un valore di φ pari a 0,58; anche per la seconda occorrenza del verbo viene scelto lo stesso significato;

-> verbo marry, significato scelto: `take in marriage` con un valore di φ pari a 0,39;

-> verbo see, significato scelto: `perceive by sight or have the power to perceive by sight` con un valore di φ pari a 0,82.

- Frase:

In the day time the street was dusty but at night the dew settled the dust and the old man liked to sit late because he was deaf and now at night it was quiet and he felt the difference.

Anche in questa frase sono presenti più verbi; l'algoritmo dà i seguenti risultati:

->verbo be, significato scelto (per tutte le occorrenze del termine): `have the quality of being; (copula, used with an adjective or a predicate noun),` con valore di φ pari a 0,59;

->verbo settle, significato scelto: `become settled or established and stable in one's residence or life style,` con valore di φ pari a 0,54;

->verbo like, significato scelto: `prefer or wish to do something,` con valore di φ pari a 0,51;

->verbo sit, significato scelto: `sit around, often unused,` con valore di φ pari a 0,43;

5.4 Efficienza del modulo di word sense disabiguation

In questa tesi si è preferito lavorare sull'efficacia degli algoritmi di word sense disambiguation piuttosto che sulla loro efficienza. Questa infatti risulta molto scarsa per diversi motivi. Primo fra tutti l'utilizzo del parser MontyTagger, il quale è molto accurato dal punto di vista dell'adeguatezza dei risultati, ma non loè altrettanto per quanto riguarda la velocità di esecuzione.

Inoltre, soprattutto nel caso dell'algoritmo dei verbi vengono fatti moltissimi accessi alle gerarchie di WordNet: per ogni significato di un verbo bisogna estrarre i nomi presenti nella definizione e ognuno di essi va confrontato con tutti i nomi presenti nella frase originale; ognuno di questi confronti a sua volta implica la ricerca all'interno di WordNet del concetto minimo comune ai due termini.
Come per i test di efficacia, anche per l'efficienza sono state usate le collezioni di dati presentate nel paragrafo 5.1. Per ognuna di esse è stata valutata l'efficienza nei casi in cui il parametro *primaedopo* assuma i valori 0,1 e 2.
Nel grafico di figura 5.4 vengono riportati i tempi medi di esecuzione per frase delle tre collezioni al variare del parametro *primaedopo.*

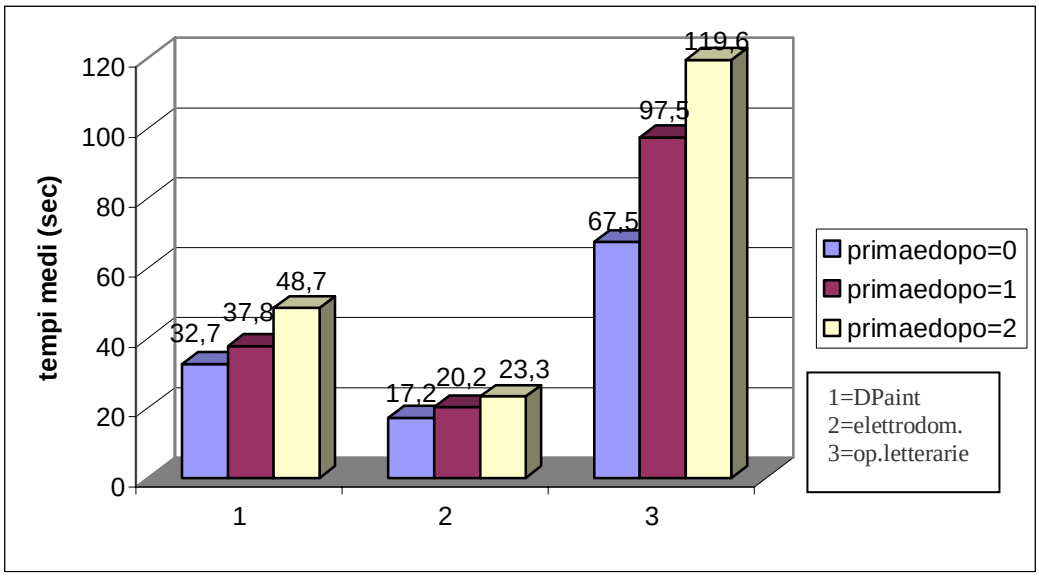


Fig. 5.4 Tempi medi di esecuzione

Come si può notare le due collezioni relative ai manuali tecnici hanno tempi di esecuzione migliori, ciò è dovuto al fatto che sono prevalentemente composti da frasi brevi e perciò con pochi termini da analizzare. Nel caso invece della collezione di tipo letterario i tempi di esecuzione sono decisamente peggiori: ciò è dovuto al fatto che le frasi sono più lunghe e contengono perciò un maggior numero di termini da confrontare.

5.5 Conclusioni

Mediante questo progetto sono stati raggiunti diversi obiettivi:

- ✓ è stata definita una funzione di similarità semantica che desse una misura della somiglianza di significato tra due concetti;
- ✓ mediante questa funzione sono stati sviluppati gli algoritmi di word sense disambiguation e si è dato, ad ogni significato di ogni termine considerato, un "peso" che indicasse la confidenza con cui poter giudicare quanto un significato fosse più appropriato di un altro. In questo modo si è associato ad ogni parola il significato che meglio le si addice.
- ✓ si è poi aumentata l'efficacia dell'algoritmo di word sense disambiguation relativo ai nomi valutando anche i nomi delle frasi adiacenti.

Per quanto riguarda gli sviluppi futuri si potrebbero potenziare gli algoritmi per la risoluzione di ambiguità semantiche nei seguenti modi:

- ✓ per potenziare il word sense disambiguation dei verbi si potrebbe valutare l'idea di analizzare le frasi in base all'analisi logica: in questo modo si determinano soggetto ed eventuale complemento oggetto di un verbo. Confrontando poi la struttura della frase con quella degli esempi d'uso dati da WordNet si potrebbero valutare solo i significati con lo stesso tipo di struttura.
- ✓ per quanto riguarda l'algoritmo di word sense disambiguation per i nomi si può valutare la possibilità di confrontare, oltre agli iperonimi, i meronimi o gli olonimi dei sostantivi.

Appendice A

Il linguaggio Java

Per rendere più comprensibile il codice riportato nell'appendice B, vengono riportate le caratteristiche principali del linguaggio Java.

A.1 Caratteristiche del linguaggio Java

Il linguaggio di programmazione e l'ambiente Java sono stati sviluppati dai ricercatori della Sun Microsystems al fine di risolvere i più comuni problemi della programmazione moderna, primi fra tutti la semplicità e la portabilità.

Java è un linguaggio di programmazione orientato agli oggetti (OOL) che si ispira al C++.

Java dà la possibilità di creare applicazioni (applet) in grado di essere richiamate ed eseguite da apposite applicazioni client del World Wide Web. E' stata proprio quest'ultima caratteristica, a rendere Java subito molto popolare.

Un altro motivo del successo di Java è l'indipendenza dalla piattaforma, cioè la capacità di un programma di poter essere eseguito su sistemi diversi.

A differenza del C++, in un programma Java tutte le istruzioni sono contenute nelle classi, le quali, una volta compilate, costituiscono ognuna un modulo a sé stante. Inoltre in Java non esiste la possibilità di accedere direttamente alla memoria. Non sono previsti né puntatori, né funzioni di allocazione e deallocazione della memoria. Questa caratteristica, rende meno flessibile il linguaggio, ma allo stesso tempo sicuro ed affidabile.

In Java l'impossibilità di manipolare direttamente la memoria rende più affidabili i programmi. Infatti, la maggior parte degli errori che si verificano in fase di esecuzione in un programma scritto in C o C++ sono dovuti ad accessi incorretti alla memoria eseguiti dai puntatori e dalle funzioni di allocazione e deallocazione

della memoria. In Java, invece, la gestione dell'allocazione e della deallocazione della memoria è gestita automaticamente (automatic garbage collection).
Infine, attraverso i metodi nativi, un programma Java può includere programmi scritti in C e C++, dando la possibiltà di usare vecchio software sviluppato con questi linguaggi. I metodi nativi in Java forniscono pure un mezzo che consente di accedere direttamente alla memoria nei casi in cui ciò si renda indispensabile.

A.2 Strumenti del linguaggio Java

In questo paragrafo verranno descritti gli elementi specifici del linguaggio Java che forniscono supporto alla programmazione ad oggetti: le classi, i package e le interfacce.

A.2.1 Le classi

Una classe è un modello che definisce un tipo di oggetto. Una classe incorpora le caratteristiche di una famiglia specifica di oggetti e ne fornisce una rappresentazione generale, mentre un'istanza ne è la rappresentazione concreta.
Nella stesura di un programma Java, si progetta e si costruisce una famiglia di classi, ossia si definiscono oggetti ai quali associamo delle funzionalità che specificano quali comportamenti possono avere.
Una classe è composta essenzialmente da due parti: gli attributi (variabili istanza e di classe) e il comportamento (metodi istanza e di classe).

A.2.1.1 Variabili istanza

Le variabili istanza definiscono gli attributi di un oggetto. La classe definisce il tipo dell'attributo, ogni istanza contiene il proprio valore dell'attributo.
Una variabile si considera d'istanza se dichiarata al di fuori dalle definizioni dei metodi, essa può essere impostata al momento della creazione dell'oggetto oppure usata come ogni altra variabile e modificata durante l'esecuzione del programma.

A.2.1.2 Variabili di classe

Le variabili di classe sono visibili a tutta una classe e a tutte le sue istanze. Spesso sono usate per mantenere traccia di uno stato globale, relativo ad una famiglia di oggetti, oppure per le comunicazioni tra istanze della stessa classe.
La dichiarazione di variabili di classe avviene premettendo la parola *static* alla normale dichiarazione.

A.2.1.3 Metodi istanza

Un metodo istanza è una funzione definita all'interno di una classe, che può agire sulle istanze della classe. Gli oggetti comunicano fra loro attraverso i metodi: una classe può chiamare i metodi di un'altra per comunicarle cambiamenti avvenuti nell'ambiente, o per chiederle di modificare il proprio stato.
La definizione di un metodo è costituita dalle seguenti parti: nome del metodo, tipo di oggetto o di dato del valore che il metodo restituisce, elenco dei parametri in ingresso e corpo del metodo. Nel caso un metodo ritorni un valore, è necessario usare l'istruzione *return*, che provvede alla restituzione del risultato.
Il passaggio dei parametri avviene per valore per i tipi primitivi, gli oggetti sono invece passati per riferimento.

A.2.1.4 Metodi di classe

I metodi di classe sono disponibili per ogni istanza della classe e possono essere messi a disposizione di altre classi, possono perciò essere richiamati indipendentemente dall'esistenza di un'istanza della classe relativa.
Come per le variabili, un metodo di classe si definisce attraverso l'uso della clausola *static*.
Come regola generale, i metodi che operano su un oggetto specifico dovrebbero essere definiti come metodi di istanza, mentre i metodi che forniscono funzioni di utilità generica, ma non agiscono direttamente sulle istanze, possono ricoprire il ruolo di metodi di classe.

A.2.1.5 Metodi costruttori

La definizione di una classe può contenere, oltre ai metodi normali, dei metodi definiti *costruttori*. Tali metodi non possono essere chiamati direttamente, come per i metodi normali, ma vengono richiamati automaticamente da Java.

Quando, mediante l'operatore *new*, si crea una nuova istanza di una classe, java compie diversi passi, fra i quali invocare il metodo costruttore. Nel caso di una classe che non disponga di costruttori, l'oggetto viene comunque creato, ma può essere necessario impostare le variabili istanza, oppure invocare altri metodi che inizializzino l'oggetto.

Grazie ai costruttori è possibile impostare i valori delle variabili istanza e inizializzare l'oggetto in base a quei valori. Per una classe, si possono definire più costruttori omonimi, che si differenziano per il numero e il tipo dei parametri.

La differenza fondamentale tra i costruttori e gli altri metodi è che il costruttore ha sempre lo stesso nome della classe e non restituisce un risultato.

A.2.1.6 Metodi conclusivi

Il metodo conclusivo è invocato immediatamente prima che un oggetto venga eliminato e la sua memoria liberata. La classe *Object* definisce un metodo conclusivo che non esegue nessuna operazione. Per definire un metodo conclusivo per una qualsiasi classe bisogna ridefinire il metodo *finalize()*.

I metodi conclusivi sono normalmente utilizzati per ottimizzare la rimozione di un oggetto: ad esempio per liberare risorse esterne o per cancellare riferimenti ad altri oggetti.

A.2.1.7 Variabili locali

Nelle definizioni dei metodi si possono dichiarare variabili che abbiano importanza solo al loro interno. Le variabili locali si possono utilizzare anche all'interno dei blocchi (la variabile non è visibile fuori dal blocco di istruzioni). In ogni caso, al termine dell'esecuzione del metodo o del blocco, le sue variabili locali vengono eliminate e di conseguenza i loro valori perduti.

A.2.1.8 Costanti

É possibile definire costanti mediante l'uso della parola chiave *final* che va anteposta alla dichiarazione del del tipo della costante.
In Java si possono creare costanti solo per le variabili istanza e di classe, non per quelle locali.

A.2.2 Le interfacce

Le interfacce sono classi astratte lasciate completamente senza implementazione. Ciò significa che nelle classi non è stato implementato alcun metodo.
I vantaggi nell'uso delle intefacce sono gli stessi offerti dall'utilizzo delle classi astratte. Le interfacce costituiscono un mezzo per definire protocolli per una classe, svincolandosi completamente dai dettagli dell'implementazione.
Una classe può implementare diverse interfacce, simulando così il concetto di ereditarietà multipla, con questo sistema però è possibile ereditare soltanto le descrizioni dei metodi, non la loro implementazione.

A.2.3 I package

I *package* vengono utilizzati per classificare e raggruppare le classi. Nel caso un'istruzione package compaia in un file sorgente Java, deve trovarsi all'inizio del file, prima di ogni altra istruzione: in questo modo tutte le classi dichiarate in quel file saranno raggruppate insieme.
E' possibile strutturare ulteriormente i package in una gerarchia simile, sotto certi aspetti, alla gerarchia ereditaria, dove ogni livello rappresenta un gruppo più ristretto e specifico di classi. Anche la libreria di classi di Java ha questo tipo di organizzazione.
Grazie all'utilizzo dei package si possono risolvere conflitti tra nomi di classi create da diversi soggetti in tempi diversi.
Ogni volta che nel proprio codice Java si fa riferimento ad una classe attraverso il suo nome, si utilizza un package. Quando si usano classi di un package diverso da quello in cui si sta lavorando, bisogna far precedere al nome della classe quello del package che la contiene, formando così un riferimento unico. Questo

procedimento può rivelarsi scomodo, soprattutto nel caso di package con nomi molto lunghi; per ovviare a ciò è consentito "importare" i nomi delle classi desiderate nel proprio programma, ed in seguito è possibile riferirsi ad esse senza alcun prefisso.
Tutte le istruzioni di *import* devono trovarsi dopo l'istruzione *package*, ma prima di ogni definizione di classe. L'importazione avviene attraverso l'istruzione:
`import <nome_package>.<nome_classe>`.
Per consentire l'utilizzo di tutte le classi di un package, si pone l'asterisco al posto di `<nome_classe>;` in questo modo è consentito l'accesso alle classi del package ma non a quelle dei sottopackage.

A.2.4 Alcune parole chiave

In questo paragrafo vengono descritte le parole chiave più significative del linguaggio Java.

extends: è usata nelle definizioni delle classi per indicare che la classe che si sta generando, eredita da un'altra classe ossia è una sottoclasse.

new: quando si crea un nuovo oggetto, si usa *new*, seguito dal nome dalla classe di cui si vuole creare un'istanza e dalle parentesi tonde. Le parentesi non possono essere omesse, anche nel caso siano vuote. All'interno di esse è possibile specificare parametri che saranno trattati dai metodi costruttori.
Ogni volta che si usa la *new*, Java esegue una sequenza di tre operazioni:

- alloca la memoria per l'oggetto;
- inizializza le variabili istanza dell'oggetto ai valori iniziali indicati, se non viene indicato niente, le variabili vengono inizializzate con i valori di default (ad esempio 0 per gli int, null per gli oggetti, false per i boolean, ecc.);
- invoca il metodo costruttore della classe.

this: questa parola chiave si riferisce all'oggetto corrente, e può trovarsi ovunque si possa inserire un riferimento ad un oggetto;

L'omissione di *this* per le variabili istanza è possibile se non esistono variabili locali con lo stesso nome. Dato che *this* è un riferimento all'istanza corrente di una classe, lo si può utilizzare solo nel corpo di un metodo istanza. I metodi di classe non possono usare *this*.

Ci sono un gruppo di parole chiave che vengono utlizzate per la gestione delle eccezioni; qui ne viene riportata una breve descrizione, per una spiegazione più dettagliata si rimanda al paragrafo A.6.
try: questa parola chiave indica che il blocco che la segue va trattato in modo speciale; in particolare il codice all'interno del blocco può catturare le eccezioni;
catch: il blocco catch gestisce le eccezioni catturate nel blocco try;
finally: il codice contenuto in questo blocco viene sempre eseguito, qualsiasi sia il modo in cui si è usciti dal blocco try.

A.3 La classe "Vector"

Alcuni tipi di strutture dati, come gli elenchi a collegamento dinamico e le code richiedono, nella loro implementazione, l'utilizzo dei puntatori. In Java non è possibile crearle direttamente perché i puntatori non esistono. Per sopperire a questa mancanza è fornita la classe Vector.
Rispetto all'uso diretto dei puntatori, la classe Vector presenta vantaggi e svantaggi. Ad esempio contribuisce a mantenere semplice il linguaggio e leggibile il codice, limita però i programmatori nell'utilizzo dei programmi più sofisticati.
Per ottimizzarne l'utilizzo, l'oggetto Vector dispone di diverse proprietà che si possono impostare. Per esempio si può decidere la dimensione della capacità prima di inserire un gran numero di oggetti, riducendo così la necessità di riallocare l'elenco per incrementarlo. Anche la capacità di memorizzazione iniziale e l'incremento della capacità possono essere specificate nel costruttore. La capacità è aumentata in modo automatico, è però possibile aumentarla di un numero minimo di elementi per volta.

Per aggiungere elementi ad un Vector si usano due metodi: *addElement()*, che aggiunge l'elemento alla fine del Vector, e *insertElementAt()*, che invece inserisce l'elemento nella posizione specificata.
E' possibile accedere agli elementi di un Vector attraverso vari metodi: con *elementAt()* si indica l'elemento la cui posizione è indicata come parametro, *firstElement()* si posiziona sul primo elemento del Vector e *lastElement()* sull'ultimo.
Un altro metodo di accesso si ottiene mediante l'uso dell'oggetto *Enumeration*. Esso è un'interfaccia che implementa una serie di metodi utili per enumerare un elenco. Un oggetto che implementa questa interfaccia può essere usato per iterare un elenco una sola volta, in quanto l'oggetto Enumeration si consuma nell'utilizzo.
L'interfaccia Enumeration ha due soli metodi: *hasMoreElements()*, che restituisce *true* se l'oggetto ha altri elementi, e *nextElement()* che restituisce il prossimo elemento dell'oggetto Enumeration.
Ne seguente codice:

```
for (Enumeration e = v.elements(); e.hasMoreElements(); )
 { System.out.println(e.nextElement()); }
```

si vede come viene usato l'oggetto Enumeration per accedere agli elementi di un Vector(v). L'oggetto Enumeration viene creato dal metodo *elements()* di Vector.

A.4 Modificatori di accesso

Il modificatore di accesso di default specifica che soltanto le classi all'interno dello stesso package abbiano accesso alle variabili e ai metodi di una classe. In altre parole i membri di una classe con accesso predefinito hanno visibilità limitata alle altre classi dello stesso package.
In assenza di altri modificatori di accesso viene applicato automaticamente quello predefinito, non esistono parole chiave per indicarlo.

A.4.1 Public

Il modificatore di accesso *public* specifica che le variabili e i metodi di una classe sono accessibili a chiunque, sia all'interno che all'esterno della classe. I membri *public* hanno quindi visibilità globale, qualsiasi altro oggetto può accedervi.

A.4.2 Protected

Questo modificatore di accesso indica che i membri di una classe sono accessibili solo ai metodi di quella classe e a quelli delle relative sottoclassi. Perciò i membri *protected* di una classe hanno visibilità limitata alle sottoclassi.

A.4.3 Private

Il modificatore di accesso più restrittivo è *private*, esso specifica che i membri di una classe sono accessibili solamente alla classe in cui sono definiti. In questo caso perciò, nessuna altra classe, sottoclassi incluse, ha accesso ai membri *private* di una classe.

A.4.4 Synchronized

Questo modificatore viene usato per specificare che un metodo è un *thread protetto*, ossia che in un metodo *synchronized* è concesso un solo percorso di esecuzione alla volta. In un ambiente multithreading come quello di Java, è normalmente possibile che nello stesso codice vengano eseguiti in contemporanea diversi percorsi di esecuzione. Attraverso il modificatore *synchronized* si permette ad un solo thread di accedere al metodo in un dato momento, tutti gli altri thread dovranno attendere il loro turno.

A.4.5 Native

Si usa questo modificatore per identificare i metodi con implementazione nativa, cioè implementazione scritta in un linguaggio diverso. Esso comunica al compilatore che l'implementazione di un metodo si trova in un file esterno. Le

dichiarazioni di metodi *native*, sono per questo motivo diverse dalle altre: non hanno corpo.
La dichiarazione di un metodo *native* termina semplicemente con un punto e virgola, senza codice Java racchiuso tra parentesi graffe, ciò perché i metodi nativi sono implementati nel codice che risiede nei file sorgenti esterni.

A.5 Garbage Collector

Come già accennato, Java non offre al programmatore la possibilità di deallocare gli oggetti in maniera esplicita. Quando non esistono più riferimenti a un oggetto, cioè quando non ci sono più oggetti che lo utilizzano, lo si può distruggere.
Il *garbage collector* è una speciale routine di sistema che scandisce periodicamente la zona di memoria che contiene gli oggetti (Java Heap), alla ricerca di oggetti non più referenziati, per eliminarli.
Uno svantaggio derivante dall'uso di questa routine è la riduzione delle prestazioni, infatti il garbage collector è a tutti gli effetti un thread che viene eseguito in parallelo ad ogni altro programma.
Ogni algoritmo di garbage collection deve fare essenzialmente tre cose:

- determinare l'oggetto che deve essere eliminato, fase detta anche *garbage detection*;
- liberare la corrispondente zona di memoria e renderla disponibile per altri oggetti;
- mantenere la frammentazione dello heap al minimo possibile.

A.5.1 Garbage detection

L'attività di garbage detection deve determinare quali oggetti sono referenziati, per questo motivo si definiscono le *root*, o radici, che sono gli oggetti sempre presenti durante l'intera esecuzione del programma. Nel caso di un'applicazione Java, è senz'altro una root l'oggetto a cui appartiene il metodo *main()*. Quali altri oggetti saranno ritenuti root dipende dall'implementazione dell'algoritmo di garbage collection, solitamente si considerano root tutti gli oggetti che abbiano uno *scope* coincidente con il corpo del main.

Agli altri oggetti viene assegnata una proprietà detta *reachability* (raggiungibilità) dalla root. Si dice che un oggetto è raggiungibile se esiste un percorso di riferimenti che, partendo dalla root, permetta di indirizzarlo. E' evidente che gli oggetti raggiungibili sono referenziabili e quindi vivi, mentre gli altri possono essere rimossi. Oggetti raggiungibili da oggetti vivi sono anch'essi vivi.
Gli algoritmi più comunemente utilizzati per la determinazione della raggiungibilità sono due: *reference counting* e *tracing*.
Reference counting: ad ogni nuovo oggetto creato, viene associato un contatore inizializzato a 1. In ogni istante il contatore tiene conto del numero dei riferimenti all'oggetto a cui è collegato, tutte le volte che viene creato un nuovo riferimento ad un oggetto, il relativo contatore viene incrementato, mentre se al riferimento viene associato un nuovo valore o l'oggetto esce dal suo scope, il contatore viene decrementato. Il garbage collector dovrà quindi verificare se il contatore relativo a un oggetto è nullo: in questo caso l'oggetto non è referenziato, e quindi si può eliminare.
Tracing: questo algoritmo consiste nel prendere come riferimento gli oggetti root. Partendo da essi si marcano tutti gli oggetti collegati tra loro da un riferimento. Dopo questa operazione, gli oggetti che non marcati saranno quelli non raggiungibili e che possono quindi essere eliminati .

A.5.2 Tecniche di deframmentazione

Un importante compito del garbage collector è quello di mantenere entro limiti accettabili la frammentazione dello heap, conseguenza del processo di continuo alternarsi di operazioni di allocazione e deallocazione di memoria durante l'esecuzione di un programma. La frammentazione potrebbe portare a gravi conseguenze se non tenuta sotto controllo, ad esempio un oggetto potrebbe non essere allocato perché non è disponibile un gruppo locazioni contigue in grado di contenerlo.
Le tecniche di deframmentazione più comuni sono il *compacting* e il *copying*. Entrambe eseguono opportuni spostamenti degli oggetti.

Compacting: questo metodo sposta tutti gli oggetti validi verso un estremo dello heap in modo che l'altro lato sia composto dalla memoria libera per l'allocazione di nuovi oggetti. La conseguenza principale è che tutti i riferimenti agli oggetti rilocati devono essere aggiornati: ciò può essere un problema per l'efficienza; viene quindi aggiunto un livello di indirizzamento indiretto per semplificare il procedimento.

Copying: tutti gli oggetti validi vengono copiati in una nuova area di memoria, inseriti uno di seguito all'altro. La vecchia area di memoria viene così lasciata libera. Un vantaggio di questo sistema è la perfetta compatibilità con il metodo di *tracing*, infatti, quando un oggetto viene marcato, lo si può immediatamente copiare nella nuova area. Gli oggetti che non vengono copiati nella nuova area sono quelli non marcati, perciò verranno eliminati.

A.6 Gestione delle eccezioni

Le eccezioni sono uno strumento avanzato per la gestione degli errori, permettono di intercettarli e di specificare il modo di gestirli.

Un metodo che debba intercettare le eccezioni generate dai metodi che invoca, deve inserire le chiamate all'interno di un blocco *try*, quando viene generata una eccezione, ad essa viene associato il modo per gestirla attraverso un blocco *catch*.

Ad un blocco try possono essere associati più blocchi catch, uno per ogni tipo di eccezione da gestire.

Ogni volta che un metodo contenuto in un blocco *try* genera una qualsiasi eccezione, l'esecuzione viene interrotta e il controllo passa al relativo blocco *catch*. Se questo è in grado di gestire l'eccezione prosegue l'esecuzione, altrimenti passa l'eccezione a chi ha invocato il metodo. Questo processo continua fino a che un blocco catch in grado di gestirla intercetta l'eccezione, o finché questa arriva al metodo *main()* senza essere stata intercettata. In questo caso l'applicazione viene terminata.

É inoltre possibile creare eccezioni proprie che verranno gestite come quelle normali.

A.6.1 Blocchi catch multipli

Può accadere che un metodo debba intercettare diversi tipi di eccezioni. In Java è possibile utilizzare più blocchi *catch*, uno per ogni tipo di eccezione che si vuole gestire.

Ogni volta che una eccezione è catturata in un blocco try, essa viene intercettata e gestita dal primo *catch* del tipo appropriato. Per ogni serie di blocchi *catch* ne viene eseguito sempre solo uno. I blocchi *catch* assomigliano alle dichiarazione dei metodi.

A.6.2 La clausola finally

Java introduce un nuovo concetto per la gestione delle eccezioni: la clausola *finally*. Essa viene posta prima di un blocco di codice che verrà comunque eseguito, sia che venga generata un'eccezione sia in caso contrario.

Il blocco *finally* viene eseguito subito dopo il blocco *try*, oppure, nel caso di eccezioni, dopo che il *catch* ha avuto modo di gestirle.

Appendice B

Codice Java

Package wsd

B.1 Classe Elaborazioni

```
package wsd;

import java.io.*;
import java.sql.*;
import java.util.*;
import java.lang.Math.*;
import montytagger.JMontyTagger;
import edu.brandeis.cs.steele.wn.*;
import wsd.risultati;

public class Elaborazioni {

   public static Vector struttura = new Vector(0,0);
   public static Vector struttura2 = new Vector(0,0);
   public static int livello = -1, indice=0;


/**
 * Estrae 'simbolo' da una stringa del tipo "parola/simbolo"
 * contenuta in un vettore
 */
  public static int cercaLivello(Vector v, int indice) {

    String riga = v.elementAt(indice).toString();
    int slash = riga.indexOf("/");
    String parola = riga.substring(slash + 1, riga.length());
    Integer l = new Integer(parola);
    int livello = l.intValue();
    return livello;

  }
```

```
 /**
  * Estrae 'parola' da una stringa del tipo "parola/simbolo"
  * contenuta in un vettore
  */
 public static String cercaParola(Vector v, int indice) {

   String riga = v.elementAt(indice).toString();
   int slash = riga.indexOf("/");
   String parola = riga.substring(0,slash);
   return parola;
 }

/**
 * Verifica se la stringa 'ip' è già stata inserita nel vettore 'v'
 */
 public static boolean cercaParola(Vector v,String ip) {
   boolean ok = false;

   for(int i=0;i<v.size();i++) {
     risultati r = (risultati) v.elementAt(i);
     if (r.getIpernimo().equalsIgnoreCase(ip) == true) {
       ok = true;
       break;
     }

   }
   return ok;
 }

/**
 * cerca la posizione di parola all'interno di frase a partire dalla posizione inizio
 */
 public static int cercaPosiz(String parola, String frase, int inizio) {

   int i, posiz = 0;

   StringTokenizer st = new StringTokenizer(frase, " \t\n\r\f,:.;!?");

   while(st.hasMoreTokens()) {
     String token = st.nextToken();
     posiz++;
     if (posiz >= inizio)
       if (token.equalsIgnoreCase(parola) == true) break;

       }

   return posiz;
 }
```

```
/**
 * Cerca un iperonimo comune a 2 gerarchie
 */
 public static String concettoComune(Vector s1, Vector s2) {
  int i, j, k, y;
  String concetto = "";

  for(i=0;i<s1.size();i++) {
   String concetto1 = cercaParola(s1, i);

   for(j=0;j<s2.size();j++) {
    String concetto2 = cercaParola(s2, j);
    if ((concetto1.equalsIgnoreCase("")==true)||(concetto2.equalsIgnoreCase("")==true))
        break;

    if (concetto1.equalsIgnoreCase(concetto2) == true) {
     concetto = concetto1;
     break;
    }
   }
   if (concetto.equalsIgnoreCase("") == false) break;

  }

  return concetto;

 }

/**
 * Cerca se esiste un iperonimo comune a "parola1" e "parola2"
 */

  public static Vector confrontaStringhe(String parola1, String parola2,
        DictionaryDatabase dict) {

  int num1, num2, p, q=0, numsalti, nums1, nums2, s, min = 100, minprec;
  String ip_comune = "", ip_minimo = "";
  IndexWord word1= null, word2 = null;
  Vector ris = new Vector(0,0);
  wsd.risultati r;


  word1 = dict.lookupIndexWord(POS.NOUN, parola1);
  word2 = dict.lookupIndexWord(POS.NOUN, parola2);

  if ( (word1 != null) && (word2 != null)) {
   Synset[] senses1 = word1.getSenses();
```

```
Synset[] senses2 = word2.getSenses();
num1 = senses1.length;
num2 = senses2.length;

for (p = 0; p < num1; p++)
  for (q = 0; q < num2; q++) {
     //si cercano gli ipernimi e ne viene memorizzato il livello
     hypernyms(senses1[p], PointerType.HYPERNYM, "concetti1");
     indice = 0;
     livello = -1;
     hypernyms(senses2[q], PointerType.HYPERNYM, "concetti2");
     indice = 0;
     livello = -1;
     //ordino le gerarchie
     ordina(gerarchia);
     ordina(gerarchia2);
     ip_comune = concettoComune(gerarchia, gerachia2);

     if (ip_comune.equalsIgnoreCase("") == false) { //esiste un concetto comune
       nums1 = contaSalti(gerarchia, ip_comune);
       nums2 = contaSalti(gerarchia2, ip_comune);
       numsalti = nums1 + nums2;
       minprec = numsalti;
       if (minprec < min)  {
         min = minprec;
         }
       ris.addElement(new risultati(ip_comune, numsalti));
       }
       gerarchia.removeAllElements();
       gerarchia2.removeAllElements();

  }


 }

 Vector ris2 = new Vector(0,0);
 for(int k=0;k<ris.size();k++) {
   r = (risultati) ris.elementAt(k);
   ip_minimo = r.getIpernimo();
   s = r.getSalti();
   if (s==min) {

     boolean ok = cercaParola(ris2, ip_minimo);
     if (ok == false)
       ris2.addElement(new risultati(ip_minimo, s));
   }
 }
```

```
     return ris2;
}

/**
 * Conta il numero di salti nella "gerarchia" c per arrivare all'iperonimo "iper"
 */

 public static int contaSalti(Vector c, String iper) {

  int i , num = 0;
  String elem;

  for (i=0;i<c.size();i++) {
   elem = cercaParola(c, i);
   if (elem.equalsIgnoreCase(iper) == true) {
     num = cercaLivello(c, i);
     num++;
     break;
   }
  }
  return num;

 }

/**
 * In base ai tag nella frase parserizzata, fa una lista dei nomi e una dei verbi
 */
  public static liste elabora(String pars, String frasiOrig, int numfrase) {

    int pos, inizio = 1, i_nomi = -1, i_verbi = -1;;
    String frase = "";
    String posiz = "";
    String parola = "", simbolo = "";
    Vector listaNomi = new Vector(0,0);
    Vector listaVerbi = new Vector(0,0);

    StringTokenizer s = new StringTokenizer(pars);
    while (s.hasMoreTokens()) {
      String riga = s.nextToken();
      int x = riga.indexOf("/");
      parola = riga.substring(0,x);
      simbolo = riga.substring(x + 1,riga.length());
      if (( simbolo.compareTo("NN") == 0 )||(simbolo.compareTo("NNS")==0))
            {
              pos = cercaPosiz(parola,frasiOrig,inizio);
              inizio = pos + 1;
```

```
            i_nomi++;
            Stemming.eng.NormalizzaFrase norm = new Stemming.eng.NormalizzaFrase();
            Stemming.eng.Output out = norm.normalizz(parola);
            String nomeSing = out.getPhrase();
            frase = frase.concat(nomeSing + " ");
            if (nomeSing.equalsIgnoreCase("") == false) { //trovato nome
              nomeSing = nomeSing.concat("/"+pos);
              listaNomi.addElement(new String(nomeSing));
            }

          }
         if (simbolo.startsWith("VB"))  // ||(simbolo.startsWith("MD"))

         {
           pos = cercaPosiz(parola, frasiOrig,inizio);
           inizio = pos + 1;
           i_verbi++;
           Stemming.eng.NormalizzaFrase norm = new Stemming.eng.NormalizzaFrase();
           Stemming.eng.Output out = norm.normalizz(parola);
           String verboBase = out.getPhrase();
           frase = frase.concat(verboBase + " ");
           verboBase = verboBase.concat("/"+pos);
           listaVerbi.addElement(new String(verboBase));

  }

}

  liste l = new liste(frase,listaNomi, listaVerbi);
  return l;

  }


/**
 * estrae la lista dei nomi e dei verbi della frase "frasiOrig"
 */
  public static liste elaboraFrasePars(String pars, String frasiOrig) throws IOException {

    int inizio = 1, pos;
    String parola = "", simbolo = "";
    Vector listaNomi = new Vector(0,0);
    Vector listaVerbi = new Vector(0,0);

     StringTokenizer s = new StringTokenizer(pars);
     while (s.hasMoreTokens()) {
        String riga = s.nextToken();
        int x = riga.indexOf("/");
```

```
        parola = riga.substring(0,x);
        simbolo = riga.substring(x + 1,riga.length());
        if (( simbolo.compareTo("NN") == 0 )||(simbolo.compareTo("NNS")==0))
            {
             pos = cercaPosiz(parola,frasiOrig,inizio);
             inizio = pos + 1;
             Stemming.eng.NormalizzaFrase norm = new Stemming.eng.NormalizzaFrase();
             Stemming.eng.Output out = norm.normalizz(parola);
             String nomeSing = out.getPhrase();
             if (nomeSing.equalsIgnoreCase("") == false) { //trovato nome
               nomeSing = nomeSing.concat("/"+pos);
               listaNomi.addElement(new String(nomeSing));

              }

            }
           if ((simbolo.startsWith("VB"))||(simbolo.startsWith("BE"))||
              (simbolo.startsWith("DO"))||(simbolo.startsWith("H")))

           {
            String verboBase;
            pos = cercaPosiz(parola, frasiOrig,inizio);
            inizio = pos + 1;
            Stemming.eng.NormalizzaFrase norm = new Stemming.eng.NormalizzaFrase();
            Stemming.eng.Output out = norm.normalizz(parola);
            verboBase = out.getPhrase();
            verboBase = verboBase.concat("/"+pos);
            listaVerbi.addElement(new String(verboBase));

  }

}

  liste l = new liste(listaNomi, listaVerbi);
  return l;

   }


/**
   * estrae la gerarchia degli iperonimi del significato "sense"
   */

public static void hypernyms(PointerTarget sense, PointerType pointerType, String c)  {

      int lung, i, j;
      String r;
```

```
    PointerTarget[] parents = sense.getTargets(pointerType);
    lung = parents.length;
    livello++;

    for (i = 0; i < lung; ++i) {

      String riga = parents[i].getDescription();
      if (c.equalsIgnoreCase("concetti1") == true) {
        r = riga;
        r = r.concat("/"+livello);
        gerarchia.addElement(new String(r));
        indice++;
      }
      else if (c.equalsIgnoreCase("concetti2") == true) {
        r = riga;
        r = r.concat("/"+livello);
        gerarchia2.addElement(new String(r));
        indice++;
      }


      hypernyms(parents[i], pointerType, c);
      livello--;
      }


   }

/**
  * ordina il vettore che contiene stringhe del tipo parola/posizione, su posizione
  */
 public static void ordina(Vector v) {

  int n, i, ordinati = 0,temp, liv_i, liv_prec;
  String stringa_i, stringa_prec;

  n = v.size();  //lunghezza del vettore
  while(ordinati < n) {
   for(i=1;i<n-ordinati;i++) {
     liv_i = cercaLivello(v, i);   //è il livello della parola alla posizione i del vettore v
     liv_prec = cercaLivello(v, i-1); //come sopra ma alla posizione i-1
     if  (liv_i < liv_prec)
     {
       stringa_i = v.elementAt(i).toString();
       stringa_prec = v.elementAt(i-1).toString();
       // scambio le righe i e i-1
       v.setElementAt(new String(stringa_i),i-1);
       v.setElementAt(new String(stringa_prec),i);
```

```
      }
     }
      ordinati++;

  }

 }

/**
 * parsing della frase
 */
  public static String parsing(String fraseOrig, JMontyTagger j) {

    String frasePars = "";
    String untagged = fraseOrig;
    frasePars = j.Tag(untagged) ;

   return frasePars;
  }


/**
 * parsing delle frasi contenute nel vettore
 */
  public static Vector parsing(Vector frasiOrig, JMontyTagger j) {

    Vector frasi_pars = new Vector(0,0);

    //JMontyTagger j = new JMontyTagger();
    for(int x=0;x<frasiOrig.size();x++) {
      String untagged = frasiOrig.elementAt(x).toString();
      String pars = j.Tag(untagged) ;
      frasi_pars.addElement(new String(pars));
    }
   return frasi_pars;
  }


/**
 * stemming delle frasi nel vettore
 */
 public static Vector stemming(Vector frasiOrig) {

    Vector frasi_stem = new Vector(0,0);

    for(int x=0;x<frasiOrig.size();x++) {
      Stemming.eng.NormalizzaFrase norm = new Stemming.eng.NormalizzaFrase();
```

```
      Stemming.eng.Output ris = norm.normalizz(frasiOrig.elementAt(x).toString());
      String frasestem = ris.getPhrase(); //contiene la frase ottenuta con lo stemming
      String posizioni = ris.getSPos();
      frasi_stem.addElement(new Stemming.eng.Output(0,frasestem,posizioni));
    }
    return frasi_stem;
  }

} // fine classe Elaborazioni
```

B.2 Classe Disambiguation

```
package wsd;

import java.util.*;
import java.io.*;
import java.sql.*;
import java.lang.Math.*;
import edu.brandeis.cs.steele.wn.*;
import montytagger.JMontyTagger;

public class Disambiguation {

  public static int D = 16;
  public static double PI = 3.14159265359;
  public static Vector lista = new  Vector(0,0);

/**
   * metodo che ritorna la lista di iperonimi del significato passato come primo parametro
   */
public static Vector ancestor(PointerTarget sense, PointerType pointerType) {

    String ip;
    PointerTarget[] parents = sense.getTargets(pointerType);

    for (int i = 0; i < parents.length; ++i) {

      ip = parents[i].getDescription();
      lista.add(new String(ip));
      ancestor(parents[i], pointerType);

     }

   return lista;
   }
```

```
/**
 * metodo che risolve l'algoritmo di word sense disambiguation dei nomi
 */

public static Vector wsdNomi(Vector listaParole, DictionaryDatabase dict) throws
IOException {

    int x, n, i, j, k, h, s, num_ip, numi, numj;
    int  maxsenses = 0;
    double sim, prec, norm_el;
    double alfa = 0.7, beta = 0.3;

    String parolai, parolaj;
    Vector nomi = new Vector(0, 0);
    Vector ris = new Vector(0, 0);
    Vector ipernimi = new Vector(0, 0);
    IndexWord word, wordi, wordj;
    risultati r;

    n = listaParole.size();
    Vector norm = new Vector(n, 0);
    Vector ris_nomi = new Vector(0,0);

    for (x = 0; x < n; x++)
    {
      String parola = elaborazioni.cercaParola(listaParole, x);
      nomi.addElement(new String(parola));
    }

    for (x = 0; x < n; x++)
    {
      word = dict.lookupIndexWord(POS.NOUN, nomi.elementAt(x).toString());
      if (word != null)
      {
        Synset[] senses = word.getSenses();
        int numsenses = senses.length;
        if (numsenses > maxsenses)
            maxsenses = numsenses;
      }
    }
  double[][] support = new double[n][maxsenses]; //riga=num parola,colonna = num sense
  double[][] phi = new double[n][maxsenses];

  // qui inizia l'algoritmo


    for (x = 0; x < n; x++)
```

```
{
  norm.addElement(new Double(0.0));
}

RandomAccessFile fileRis = new RandomAccessFile("risultati.txt", "rw");
long lung = fileRis.length();
fileRis.seek(lung);
fileRis.write(10);
fileRis.writeBytes("----- RISULTATI ALGORITMO_NOMI -----\n");
/* ----- primo ciclo ----- */

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i < j)
    {
      parolai = nomi.elementAt(i).toString();
      parolaj = nomi.elementAt(j).toString();
      wordi = dict.lookupIndexWord(POS.NOUN, parolai);
      wordj = dict.lookupIndexWord(POS.NOUN, parolaj);

      int posi = elaborazioni.cercaLivello(listaParole, i);
      int posj = elaborazioni.cercaLivello(listaParole, j);

      int z = posi - posj;
      //smorz = 4*N(0,2)+1-2/sqrt(2*PI)  gaussiana
      double smorz = 1 +
          (2 / java.lang.Math.sqrt(2 * PI)) *
          (java.lang.Math.exp( -java.lang.Math.pow(z, 2) / 8) - 1);

      if (wordi == null) fileRis.writeBytes(parolai +"->parola non trovata in
                                              WordNet\n");
      if (wordj == null) fileRis.writeBytes(parolaj +"->parola non trovata in
                                              WordNet\n");
      if ( (wordi != null) && (wordj != null))
      {
        Synset[] sensesi = wordi.getSenses();
        Synset[] sensesj = wordj.getSenses();
        numi = sensesi.length; //numero senses parola i-esima
        numj = sensesj.length; //numero senses parola j-esima

        ris = elaborazioni.confrontaStringhe(parolai, parolaj, dict);
        if (ris.size() > 0)
        {
          r = (risultati) ris.firstElement();
          s = r.getSalti();

          fileRis.writeBytes(parolai + ", " + parolaj + " ; num salti: " + s +
                        " ; iperonimo comune : " + r.getIpernimo()+"\n");
```

```
double arg = ( (double) s / (2 * D));
sim = -java.lang.Math.log(arg); //log base e
sim = sim * smorz;

num_ip = ris.size(); //numero di iperonimi minimi
String c;

for (k = 0; k < num_ip; k++)
{
  r = (risultati) ris.elementAt(k);
  c = r.getIpernimo();
  ipernimi.addElement(new String(c));
}

for (k = 0; k < numi; k++)
  for (h = 0; h < num_ip; h++)
  {
    // lista contiene gli iperonimi del sense k di parolai
    ancestor(sensesi[k], PointerType.HYPERNYM);
    String ip_minimo = ipernimi.elementAt(h).toString();
    if (lista.contains(new String(ip_minimo)) == true)
    {
      support[i][k] = support[i][k] + sim;
    }
    lista.removeAllElements();

  }
for (k = 0; k < numj; k++)
  for (h = 0; h < num_ip; h++)
  {
    // lista_ip contiene gli iperonimi del sense k di parolaj
    ancestor(sensesj[k], PointerType.HYPERNYM);
    String ip_minimo = ipernimi.elementAt(h).toString();
    if (lista.contains(new String(ip_minimo)) == true)
    {
      support[j][k] = support[j][k] + sim;
    }
    lista.removeAllElements();

  }
double incr = sim * num_ip;
Double p = (Double) norm.elementAt(i);
prec = p.doubleValue();
norm_el = prec + incr;
norm.setElementAt(new Double(norm_el), i);

Double q = (Double) norm.elementAt(j);
```

```
          prec = q.doubleValue();
          norm_el = prec + incr;
          norm.setElementAt(new Double(norm_el), j);

        }
        else
          fileRis.writeBytes(parolai + ", " + parolaj + " ; no ipernimi comuni\n");
        }
      ipernimi.removeAllElements();
    } //fine primo ciclo

fileRis.writeBytes("\n");

/* ---- secondo ciclo ---- */

for (i = 0; i < n; i++)
{
  parolai = nomi.elementAt(i).toString();

  fileRis.writeBytes("PAROLA " + i + " : " + nomi.elementAt(i).toString()+"\n");

  wordi = dict.lookupIndexWord(POS.NOUN, parolai);
  if (wordi != null)
  {
    Synset[] sensesi = wordi.getSenses();
    numi = sensesi.length;

    for (k = 0; k < numi; k++)
    {
      Double p = (Double) norm.elementAt(i);
      if (p.compareTo(new Double(0.0)) != 0)
      {
        phi[i][k] = support[i][k] / p.doubleValue();
        phi[i][k] = alfa * phi[i][k];
        if (numi == 1)
          phi[i][k] = phi[i][k] + beta;
        else
          phi[i][k] = phi[i][k] + beta * (1 - 0.8 * k / (numi - 1));


        fileRis.writeBytes("phi[" + i + "][" + k + "] = " + phi[i][k]+ "\n");

      }
      else
      {
        phi[i][k] = 1 / numi;
        phi[i][k] = alfa * phi[i][k];
        if (numi == 1)
```

```
          phi[i][k] = phi[i][k] + beta;
        else
          phi[i][k] = phi[i][k] + beta * (1 - 0.8 * k / (numi - 1));
        fileRis.writeBytes("phi[" + i + "][" + k + "] = " + phi[i][k]+ "\n");

      }
    }

  }
  fileRis.writeBytes("\n");
} //fine secondo ciclo


//estrazione risultato migliore

int inizio = 0;
String fraseSenses = "";
double max;

for (i = 0; i < nomi.size(); i++)
{
  parolai = nomi.elementAt(i).toString();
  inizio = i + 1;
  wordi = dict.lookupIndexWord(POS.NOUN, parolai);
  max = 0.0;
  if (wordi != null)
  {
    Synset[] sensesi = wordi.getSenses();
    numi = sensesi.length;

    for (k = 0; k < numi; k++)
    {
      if (phi[i][k] > max)
        max = phi[i][k];
    }
    for (k = 0; k < numi; k++)
    {
      if (phi[i][k] == max)
      {
        fileRis.writeBytes("valore migliore per " + parolai + ": " + phi[i][k] + "\n");
        fileRis.writeBytes("numero sense : " + k + "\n");
        fileRis.writeBytes("descrizione sense : " + sensesi[k].getGloss()+ "\n");
        String riga = parolai +"/"+sensesi[k].hashCode();
        ris_nomi.add(new String(riga));


      }
    }
```

```
      }
      fileRis.writeBytes("\n");
    }

    fileRis.close();

    return ris_nomi;

  }  //fine algoritmo()


  /**
   * Estrae le frasi di esempio del significato numero "numsense" di "verbo"
   */
public static Vector esempiUso(String verbo, int numsense) {

    Vector esempi = new Vector(0,0);

    DictionaryDatabase dict = new
          FileBackedDictionary("c:/Programmi/Wordnet/1.7.1/dict");
    IndexWord word = dict.lookupIndexWord(POS.VERB, verbo);
    if (word != null)
    {
      Synset[] senses = word.getSenses();
      char[] c = new char[1];
      c[0] = "";
      String da = "";
      da = da.copyValueOf(c);


      int a = 0; //indica la posizione da cui cominciare a cercare "; "
      String def = senses[numsense].getGloss(); //definizione del sense k di verbo
      String frase = "";

      int ultimo = def.lastIndexOf(';');
      int le = def.length();

      for(int i=0;i<=ultimo;i++) {
         if (def.charAt(i) == ';') {
           frase = def.substring(a, i);
           if (frase.startsWith(da)) frase = frase.substring(1,frase.length());
           if (frase.startsWith(" "+da)) frase = frase.substring(2,frase.length());
           if (frase.endsWith(da)) frase = frase.substring(0,frase.length()-1);
           a = i+1;
           esempi.add(new String(frase));
         }
      }
```

```
    if (ultimo != (def.length() - 1)) {
      frase = def.substring(ultimo + 1, def.length() - 1);
      if (frase.startsWith(da))
        frase = frase.substring(1, frase.length());
      if (frase.startsWith(" " + da))
        frase = frase.substring(2, frase.length());
      esempi.add(new String(frase));
    }

   }
   return esempi;

  }


/**
  * Estrae i nomi presenti nelle frasi di esempio del significato "numsense" di "verbo"
  */
public static Vector nomiUso(String verbo, int numsense,JMontyTagger j) throws
IOException {

Vector nomi_es = new Vector(0,0);

Vector esempi = esempi_uso(verbo, numsense); //contiene gli es del sense numsense di
                                                    //  verbo
Vector esempi_pars = elaborazioni.parsing(esempi, j);

Vector esempi_stem = elaborazioni.stemming(esempi);

for (int x=0;x<esempi.size();x++) {
     String esempio = esempi.elementAt(x).toString();
     String esempio_pars = esempi_pars.elementAt(x).toString();
     liste l = elaborazioni.elabora(esempio_pars, esempio, x);
     Vector nomi = l.getListaNomi();
     Vector verbi = l.getListaVerbi();
     Stemming.eng.Output out = (Stemming.eng.Output) esempi_stem.elementAt(x);
     String es_stem = out.getPhrase();
     Vector stem_pars = elaborazioni.stem_pars(es_stem, nomi, verbi);
     String es_stempars = stem_pars.elementAt(0).toString();

     List nomi_in_es = estraiNomi(es_stempars, nomi); //nomi presenti nell'esempio num x
     for (int i=0;i<nomi_in_es.size();i++) {
       String nome_es = nomi_in_es.get(i).toString();
       nomi_es.addElement(new String(nome_es));
     }

   }
```

```
    return nomi_es;
  }


/**
   * metodo che data una frase stem&pars ritorna la lista dei nomi in essa contenuti
   */
  public static List estraiNomi(String frasesp, Vector listaNomi) {

    List lista = new ArrayList();
    Vector nomi = new Vector(0,0);

    for (int x = 0; x < listaNomi.size(); x++)
     {
       String parola = elaborazioni.cercaParola(listaNomi, x);
       nomi.addElement(new String(parola));
     }

     StringTokenizer s = new StringTokenizer(frasesp);
     int conta = 0; int conta2 = 0;
     while (s.hasMoreTokens()) {
       String token = s.nextToken();

      for (int i = conta; i < nomi.size(); i++)
     {
       String nome = nomi.elementAt(i).toString();

       if (nome.equalsIgnoreCase(token) == true)
       {
         lista.add(new String(nome));
         conta++;

         break;
       }
     }
   }

   return lista;

  }

/**
  * metodo che risolve l'algoritmo di word sense disambiguation dei verbi
  */
  public static Vector wsdVerbi(Vector verbi, Vector nomi, JMontyTagger j) throws
      IOException {

       Vector ris_verbi = new Vector(0,0);
```

```
RandomAccessFile fileRis = new RandomAccessFile("risultati.txt", "rw");
long lung = fileRis.length();
fileRis.seek(lung);
fileRis.write(10);
fileRis.writeBytes("----- RISULTATI ALGORITMO_VERBI -----\n");

DictionaryDatabase dictionary = new
       FileBackedDictionary("c:/Programmi/Wordnet/1.7.1/dict");
for(int i=0;i<verbi.size();i++) {
  String verbo = elaborazioni.cercaParola(verbi,i);
  System.out.println("verbo : "+verbo);
  int posverbo = elaborazioni.cercaLivello(verbi,i);
  IndexWord word = dictionary.lookupIndexWord(POS.VERB, verbo);
  if (word != null) {
    Synset[] senses = word.getSenses();
    int num = senses.length;
    double[] pesi = new double[num];

    for(int k=0;k<num;k++) {
      double peso = peso_verbi(verbo,k,num,nomi,posverbo,dictionary,j);
      pesi[k] = peso;

      fileRis.writeBytes(verbo+" , sense num "+(k+1)+" , valore: "+peso+"\n");

      double R_succ = 1 - (0.9 * (k+1) / (num - 1));

      if (peso > R_succ) break;

    }

    double max = 0.0;
    for(int k=0;k<num;k++) {
      if (pesi[k] > max) max = pesi[k];
    }
    for(int k=0;k<num;k++) {
      if (pesi[k] == max) {

        ris_verbi.add(new String(verbo+"/"+senses[k].hashCode()));

        fileRis.writeBytes("     verbo "+verbo+" , sense migliore : "+(k+1)+"\n");
        fileRis.writeBytes("descrizione sense : "+senses[k].getGloss()+"\n");
        fileRis.write(10);
        break;
      }
    }
```

```
      }
    }
    return ris_verbi;
  }


  /**
    *calcola il peso phi del significato numero "numsense" di "verbo"
    */
public static double pesoVerbi(String verbo, int numsense, int totsense, Vector listaNomi,
int posverbo, DictionaryDatabase dict,JMontyTagger j) throws IOException {

    int n = listaNomi.size();
    double totgauss = 0.0;
    double sim = 0.0;
    double[] gauss = new double[n];
    double peso = 0.0, somma = 0.0;

    double R =(1 - 0.9 * numsense / (totsense - 1));

    //calcolo le gaussiane
    for(int i=0;i<n;i++) {
      String nome = elaborazioni.cercaParola(listaNomi,i);
      int pos_nome = elaborazioni.cercaLivello(listaNomi,i);
      int z = pos_nome - posverbo;
      // gauss = 4*N(0,2) + 1 - 2/sqrt(2*PI)
      gauss[i] = 1 +(2 / java.lang.Math.sqrt(2 * PI)) *(java.lang.Math.exp( -
           java.lang.Math.pow(z, 2) / 8) - 1);
      totgauss = totgauss + gauss[i];
    }

    //estraggo i nomi presenti negli esempi d'uso

    Vector nomi_es = nomi_uso(verbo, numsense, j);

    for(int i=0;i<n;i++) {
      String nome = elaborazioni.cercaParola(listaNomi,i);
      double max = 0.0;
      for(int k=0;k<nomi_es.size();k++) {
        String nome_es = nomi_es.elementAt(k).toString();
        Vector ris = elaborazioni.confrontaStringhe(nome, nome_es, dict);
        if (ris.size() > 0)
        {
          risultati r = (risultati) ris.firstElement();
          int s = r.getSalti();
          double arg = ( (double) s / (2 * D));
          sim = -java.lang.Math.log(arg); //log base e
          sim = sim / (-java.lang.Math.log((double)1/D));
```

```
        }
        else sim = 0.0;

        if (sim > max) max = sim;

      }
      somma = somma + (gauss[i]*max)/totgauss;

    }

    peso = R*somma;
    return peso;

  }


/**
   * metodo che produce la frase con i codici di WordNet
   */
  public static String codiciSenses(String frasesp, Vector ris_nomi, Vector ris_verbi)
          throws IOException {

    int inizio = 0,inizio2=0, i;
    String fraseSenses = "";
    Vector verbi = new Vector(0,0);
    Vector nomi = new Vector(0,0);

    DictionaryDatabase dictionary = new
          FileBackedDictionary("c:/Programmi/Wordnet/1.7.1/dict");

    RandomAccessFile fileRis = new RandomAccessFile("risultati.txt","rw");
    long lung = fileRis.length();
    fileRis.seek(lung);
    fileRis.writeByte(10);

    for(int j=0;j<ris_nomi.size();j++) {
      String nome = elaborazioni.cercaParola(ris_nomi,j);
      nomi.add(new String(nome));
    }

    for(int j=0;j<ris_verbi.size();j++) {
      String verbo = elaborazioni.cercaParola(ris_verbi,j);
      verbi.add(new String(verbo));
    }

    StringTokenizer st = new StringTokenizer(frasesp);
    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      if (verbi.contains(new String(token))== true)  {//è un verbo
```

```
      i = verbi.indexOf(new String(token),inizio2);
      inizio2 = i + 1;
      String verbo = elaborazioni.cercaParola(ris_verbi,i);
      int id = elaborazioni.cercaLivello(ris_verbi,i);
      fraseSenses = fraseSenses.concat(id + " ");
     }

     else  if (nomi.contains(new String(token))== true) {  // è un nome
      i = nomi.indexOf(new String(token),inizio);
      inizio = i + 1;
      String nome = elaborazioni.cercaParola(ris_nomi,i);
      int id = elaborazioni.cercaLivello(ris_nomi, i);
      fraseSenses = fraseSenses.concat(id + " ");

     }
     else { // è una parola che non viene riconosciuta come nome o verbo(non è nel diz)
      fraseSenses = fraseSenses.concat(token + " ");
     }
   } //fine while

   fileRis.writeBytes("FRASE : "+frasesp+"\n");
   fileRis.writeBytes("FRASE con SENSES: "+fraseSenses);
   fileRis.writeBytes("\n");
   fileRis.close();

   return fraseSenses;

  }

} //fine classe Disambiguation
```

B.3 Classe Liste

```
package wsd;

import java.util.*;

public class Liste
{
  Vector listaNomi = new Vector(0,0);
  Vector listaVerbi = new Vector(0,0);
  String frase = "";

  public Liste(Vector nomi, Vector verbi)
  {
    listaNomi = nomi;
    listaVerbi = verbi;
```

```
  }

  public Liste(String frase1, Vector nomi, Vector verbi)
  {
    frase = frase1;
    listaNomi = nomi;
    listaVerbi = verbi;
  }

  public Vector getListaNomi() {
    return listaNomi;
  }

  public Vector getListaVerbi() {
    return listaVerbi;
  }

  public String getFrase() {
    return frase;
  }
}
```

B.4 Classe Risultati

```
package wsd;

import java.util.*;

public class Risultati
{
    String ipernimo;
    int numsalti;

    public Risultati(String s, int num) {
      ipernimo = s;
      numsalti = num;
     }

   public String getIpernimo() {
    return ipernimo;
   }

  public int getSalti() {
   return numsalti;
 }
}
```

Bibliografia

[1] N. Ide, J. Veronis, “Word Sense disambiguation: The State of Art”, Computational Linguistics, vol. 24, number 1, p. 1-40 (1998).

[2] M. Mastermann, “Semantic message detection for machine translation, using an interlingua”, Internaional Conference on Machine Translation of Languages and Applied Language Analysis (1961).

[3] M. R. Quillian, “A design for an understanding machine”, Communication presented at the colloquium *Semantic problems in natural language*, King’s College, Cambridge University (1961).

[4] M. R. Quillian, “A revised design for an understanding machine”, Mechanical Translation, 7(1), 17-29 (1962).

[5] A. M. Collins, E. F. Loftus, “A spreading activation theory of semantic processing”, Psychological Review, 82(6), 407-428 (1975).

[6] J. R. Anderson, “Language, Memory and Thought”, Lawrence Erlbaum and Associates, Hillsdale, New Jersey (1976).

[7] J. R. Anderson, “ A Spreading Activation Theory of Memory”, Journal of Verbal Learning and Verbal Behaviour, 22(3), 261-95 (1983).

[8] R. A. Amsler, “The structure of the Merriam-Webster Pocket Dictionary”, Ph.D. Dissertation, University of Texas at Austin (1980).

[9] A. Michiels “Exploiting a large dictionary database”, Doctoral dissertation, Universitè de Liège (1982).

[10] J. Pustejovsky, “The generative lexicon”, The MIT Press, Cambridge, Massachussetts (1995).

[11] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, K. Miller, “Five papers on WordNet”, Cognitive Science Laboratory, Princeton University, 1993

[12] WordNet : www.cogsci.princeton.edu/ wn/

[13] C. Fellbaum, G. A. Miller, “Folk Psychology or Semantic Entailment? A Reply to Rips and Conrad.” *Psychological Review,* 1990.

[14] D. Gross, U. Fisher, G.A. Miller, “The Organization of Adjectival Meanings.’ *Journal of Memory and Language* 28: pag.92-106, 1989.

[15] R. Martoglia, EXTRA: Progetto e Sviluppo di un Ambiente per Traduzioni Multilingua Assistite. Tesi di Laurea, Università degli Studi di Modena e Reggio
Emilia, 2000/2001.

[16] F. Gavioli, Progetto ed Implementazione di un Algoritmo per Ricerca di Similarità tra Frasi. Tesi di laurea, Università degli studi di Modena e Reggio Emilia, 1999/2000.

[17] B. Santorini, “Part-of-speech tagging guidelines for the Penn Treebank Project”, Department of Computer and Information Science, University of Pennsylvania, 1990.

[18] A. Bies et al., “Bracketing Guidelines for Treebank II Style Penn Treebank Project”, University of Pennsylavnia, 1995.

[19] Penn Treebank Project : www.cis.upenn.edu/~treebank/

[20] A. Budanitsky, G. Hirst, “Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures”, Workshop on Wordnet and Other Lexical Resources, in the North American Chapter of the Association for Computational Linguistics (NAACL-2000, Pittsburgh, PA, June 2001).

[21] P. Resnik, “Disambiguating noun groupings with respect to WordNet senses”, Proceedings of the Third Workshop on Very Large Corpora, pag. 54-68, Association for Computational Linguistics, 1995.

[22] B. Eckel, “Thinking in Java” Ed. Apogeo