Università degli studi
di Modena e Reggio Emilia

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

EXTRA: Progetto e Sviluppo di un Ambiente per Traduzioni Multilingua Assistite

Riccardo Martoglia

Tesi di Laurea

Relatore:
Chiar.mo Prof. Paolo Tiberio

Controrelatore:
Chiar.mo Prof. Sonia Bergamaschi

Correlatore:
Dott. Federica Mandreoli

Anno Accademico 2000/2001

Ringraziamenti

Ringrazio il Professor Paolo Tiberio,
la Professoressa Sonia Bergamaschi
e il personale della LOGOS S.p.A.
per la disponibilità e la collaborazione
fornita. Un grazie particolare alla
Dottoressa Federica Mandreoli per
l'appoggio e la costante assistenza.

Parole chiave

Ricerca di Similarità
Edit Distance
Machine Translation
Allineamento
Translation Memory

Indice

Elenco delle figure

Introduzione

La traduzione è un'attività ripetitiva, che richiede un elevatissimo grado di attenzione e una vasta gamma di conoscenze. Per lungo tempo, il tentativo di automatizzare un compito di tale difficoltà è stato un sogno di enorme valenza sociale, politica e scientifica. Negli ultimi anni, la ricerca in questo campo ha conosciuto un crescente interesse e, grazie anche ai recenti progressi tecnologici, è oggi possibile un certo grado di traduzione automatica (o Machine Translation). Tra i tanti approcci proposti, uno dei paradigmi più promettenti è rappresentato dai sistemi per traduzioni MAHT (Machine Assisted Human Translation), ed in particolare dai sistemi EBMT (Example Based Machine Translation).

Un sistema EBMT traduce per analogia, utilizzando le traduzioni passate per tradurre altre frasi dalla lingua sorgente alla lingua destinazione. Al centro del funzionamento di questi sistemi vi sono le metodologie per la valutazione del grado di somiglianza tra le frasi. L'argomento principale della presente tesi è di affrontare le problematiche legate alla *ricerca di similarità* tra frasi applicata al contesto della traduzione multilingua.

Partendo dagli studi condotti nell'ambito dell'Information Retrieval e delle tecniche di ricerca comunemente utilizzate (basilari ma insufficienti per questo particolare problema), è stata definita una struttura teorica che permette di esprimere adeguatamente ed *efficacemente* se e quanto due frasi sono simili, basata su concetti a volte già noti ma mai utilizzati in questo campo, quale quello di *edit distance*.

Sono quindi stati progettati una serie di algoritmi che, appoggiandosi su un DBMS, permettono di effettuare questo nuovo tipo di ricerca. Siccome il volume di dati su cui lavora un traduttore è normalmente di notevoli dimensioni, si è fatto in modo che anche dal punto di vista dell'*efficienza* i risultati fossero di

alto livello, utilizzando *query* ottimizzate, anche nella struttura: ci si è avvalsi di una serie di $filtri$ in grado di ridurre il volume dei dati e delle operazioni senza per questo mai rinunciare alla correttezza e alla completezza del risultato.

Infine, per poter sfruttare appieno il potenziale di questi algoritmi e per fornire un certo numero di funzionalità aggiuntive complementari, questo lavoro si è evoluto e ampliato nel progetto EXTRA (EXample based TRanslation Assistant), un ambiente completo per il traduttore di testi, sviluppato su tecnologie Java, JDBC ed Oracle.

EXTRA è stato pensato in particolare per accelerare la traduzione di manuali di carattere scientifico/tecnico, ma è utilizzabile in molti altri ambiti; è importante sottolineare come questo strumento non voglia affatto *sostituirsi* alla figura del traduttore professionista, proponendo autonomamente traduzioni di nuovi documenti, ma anzi si appoggi al lavoro precedentemente svolto da questi al fine di *accelerare* il processo di traduzione.

Il programma è fondamentalmente uno strumento EBMT (Example Based Machine Translation), il cui compito predominante è quello di effettuare la cosiddetta *pretraduzione*: il traduttore fornisce come input i documenti da lui precedentemente tradotti (possibilmente di argomento attinente) e il nuovo documento da tradurre, il programma costruisce una base di dati di frasi già "approvate" (la cosiddetta *Translation Memory*) e da questa attinge per suggerire le traduzioni delle frasi più "somiglianti". A questo punto, al traduttore non rimane altro che utilizzare ed eventualmente modificare i suggerimenti, in modo da portare a termine il lavoro e, successivamente, inserirlo a sua volta nella memoria di traduzione.

Per poter fornire suggerimenti utili e accurati, il *modulo di ricerca di similarità* del software non si limita a considerare le frasi già tradotte e a proporle se riconosciute simili a quelle da tradurre: esso svolge un lavoro molto più minuzioso e innovativo, ricercando e proponendo anche le eventuali *sottoparti* simili, complete della relativa traduzione, adeguatamente estratta dal contesto globale della frase tradotta. Per poter ottenere questi risultati sono stati studiati e implementati anche una serie di algoritmi che, partendo dai due documenti forniti (uno la traduzione dell'altro), si occupano di *allineare* mediante varie elaborazioni ed in modo completamente *automatico* ed *indipendente dalle lingue* prima

le frasi all'interno dei paragrafi, poi le parole all'interno delle frasi (*modulo di allineamento*). Completano EXTRA il *modulo di stemming*, sviluppato originariamente in [13], ottimizzato ed esteso per essere inserito nel nuovo contesto e per potenziare ulteriormente la ricerca, e un'*interfaccia grafica*, progettata per rendere comodamente richiamabili e configurabili tutte le funzionalità offerte dal sistema.

Il progetto è stato svolto in collaborazione con LOGOS S.p.A., leader mondiale nella traduzione multilingua di documenti, le cui soluzioni si avvalgono di team di localizzazione professionali distribuiti in tutto il mondo.

Per quanto riguarda la struttura della tesi, essa è organizzata come segue. Nella **Prima Parte** vengono introdotti i numerosi argomenti di ricerca studiati per affrontare la realizzazione del software EXTRA. In particolare, nel **Capitolo 1** vengono descritte le principali nozioni di *Information Retrieval*, nonché lo stato dell'arte in questo campo con particolare riferimento alla ricerca di somiglianza. Inoltre, nel **Capitolo 2** vengono descritte le problematiche di ricerca sulla traduzione di testi, analizzando i vari paradigmi presenti, lo stato dell'arte dei sistemi EBMT e degli argomenti scientifici attinenti (ad esempio l'*allineamento* di corpus bilingui). Vengono anche brevemente descritti i principali software di traduzione assistita presenti in ambito commerciale.

Nella **Seconda Parte** della tesi viene descritto l'approccio seguito nella ricerca di somiglianza tra le frasi (**Capitolo 3**) e nella progettazione delle rimanenti funzionalità di EXTRA (quali gli allineamenti o l'interfaccia, **Capitolo 4**); nel **Capitolo 5** viene mostrato il vero e proprio progetto del software nel suo insieme, con Package e Class Diagram. Vengono inoltre proposti i Data Flow Diagram per meglio capirne il funzionamento. Infine, nel **Capitolo 6** vengono descritti i test e le collezioni progettati per verificare le funzionalità del software, oltre ovviamente ai risultati conseguiti.

Nelle **Appendici** vengono riportati, per completezza e come ulteriore riferimento, le parti più significative del codice JAVA ed il codice SQL relativo alle query e agli script utilizzati.

Parte I

Introduzione al problema

Capitolo 1

Analisi dell'information retrieval

1.1 Introduzione all'information retrieval

L'*Information Retrieval* (IR) riguarda classicamente la rappresentazione, la registrazione, l'organizzazione e l'accesso agli oggetti di informazione [3]. La rappresentazione e l'organizzazione di questi oggetti fornisce all'utente un facile accesso all'informazione cui è interessato. Sfortunatamente non è sempre facile caratterizzare con esattezza il bisogno di informazioni di un utente: questi deve prima tradurre questo suo bisogno in una *query* che possa essere elaborata da un motore di ricerca (o sistema di IR). Nella forma più comune, questa traduzione mantiene un set di *parole chiave* (*keyword* o *termini indice*) che riassumono la descrizione del bisogno dell'utente. Data la query dell'utente, la principale finalità di un sistema di IR è recuperare l'informazione che può essere utile o rilevante.

È importante mettere in evidenza un concetto chiave: la differenza tra *Information Retrieval* e *Data Retrieval*: quest'ultimo, nel contesto di un sistema di IR, consiste principalmente nel determinare quali documenti di una collezione contengono le parole chiave della query, cosa che non è sufficiente a soddisfare il bisogno di informazioni di un utente. Infatti, all'utente di un sistema di IR interessa più recuperare *informazioni* su un particolare argomento piuttosto che recuperare *dati* che soddisfano una particolare query. Lo scopo di un linguaggio di Data Retrieval è di recuperare tutti gli oggetti che soddisfano delle condizioni definite con chiarezza, come quelle di una espressione regolare o di algebra relazionale. Così, per un sistema di Data Retrieval, il recupero di un oggetto errato tra migliaia rappresenta un completo fallimento. Per l'Information Retrieval le cose stanno diversamente: non ci sono delle certezze assolute, gli oggetti

recuperati possono anche essere inaccurati ed è facile che piccoli errori passino inosservati. Questo poiché l'IR quasi sempre deve fare i conti con testo in linguaggio naturale che non sempre è ben strutturato e può essere semanticamente ambiguo.

Per essere veramente efficace, un sistema di IR deve in qualche modo 'interpretare' il contenuto degli oggetti informativi (i documenti) in una collezione e ordinarli in base al grado di rilevanza con la query. Questa 'interpretazione' del contenuto del documento riguarda l'estrazione di informazioni sintattiche e semantiche dal testo del documento e la loro utilizzazione per soddisfare i bisogni dell'utente. La difficoltà sta non solo nel decidere come estrarre le informazioni ma anche come usarle per stabilirne la rilevanza. Pertanto, la nozione di *rilevanza* è al centro dell'information retrieval: lo scopo primario di un sistema di IR è proprio recuperare tutti i documenti rilevanti rispetto alla query dell'utente, ordinandoli (*ranking*) rispetto a una *misura di similarità* e riducendo il più possibile il numero di documenti non rilevanti recuperati.

Negli ultimi 20 anni, l'area dell'information retrieval è andata ben oltre gli scopi primari di indicizzare testo e ricercare i documenti utili di una collezione. Oggi, la ricerca nell'IR include modellazione, classificazione e categorizzazione di documenti, architetture di sistema, interfacce utente, visualizzazione di dati, filtraggio, ecc. Inoltre, con l'introduzione del World Wide Web, l'IR ha perso la connotazione di piccola area di interesse per pochi eletti, come bibliotecai od esperti di informazione, ed ha assunto un'importanza e una rilevanza globali.

1.2 I concetti alla base dell'IR

L'efficacia del recupero delle informazioni è direttamente influenzata da due aspetti chiave: il modo in cui l'utente svolge il proprio compito (normalmente specificare un insieme di parole che esprima adeguatamente la semantica della propria richiesta di informazione) e la *vista logica* dei documenti adottata dal sistema di IR. Proprio quest'ultimo aspetto merita una spiegazione più dettagliata.

I documenti in una collezione sono spesso rappresentati da un insieme di *keyword*, le quali possono essere estratte direttamente dal testo del documento o possono essere esplicitamente indicate in un sunto iniziale preparato da uno specialista. Comunque vengano estratte, sono proprio queste parole a fornire

una *vista logica* del documento, che può essere più o meno dettagliata a seconda della tipologia.

I moderni computer stanno rendendo possibile rappresentare un documento con l'intero insieme delle sue parole. In questo caso, si dice che il sistema di IR adotta una vista (o rappresentazione) logica dei documenti di tipo *full text*. Con collezioni molto voluminose, tuttavia, persino i moderni elaboratori devono poter ridurre l'insieme delle parole rappresentative. Questo può essere ottenuto mediante *stemming* (o *normalizzazione*, che riduce diverse parole alla loro comune radice grammaticale) (vedi sezione 1.6.1) o con l'identificazione di *gruppi di nomi* (che elimina aggettivi, avverbi e verbi). Queste elaborazioni vengono dette *text operation* [3] (sezione 1.6) e, in generale, riducono la complessità della rappresentazione del documento, permettendo di passare dalla vista logica *full text* a quella dell'insieme delle parole chiave (figura 1.1).

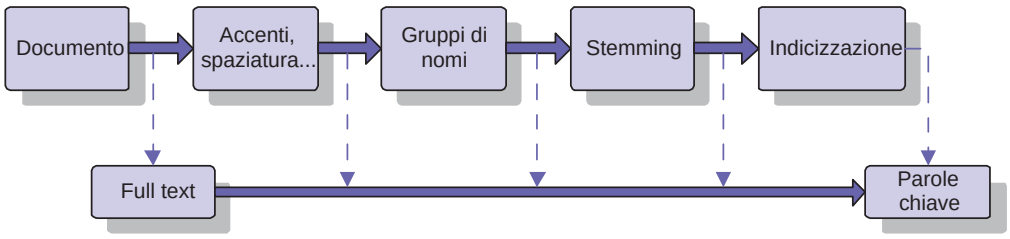


Figura 1.1: La vista logica di un documento: da full text a parole chiave

Una volta definita la vista logica dei documenti, il sistema di IR costruisce un *indice* del testo: l'indice è una struttura dati critica poiché permette una ricerca veloce su grandi volumi di dati. Possono essere utilizzate differenti strutture, ma la più popolare è il cosiddetto *inverted index* (sezione 1.7): questo contiene in buona sostanza l'elenco delle parole di interesse presenti nei vari documenti, seguite da una serie di link ai documenti stessi. Le risorse (di tempo e di spazio) spese per definire un database dei testi e per costruire gli indici sono ammortizzate dalle numerose query rivolte al sistema.

Dopo aver indicizzato il database dei documenti, la ricerca può iniziare. L'utente fornisce la propria richiesta, la quale viene analizzata da un parser e trasformata con le stesse *text operation* applicate ai testi. La query viene dunque processata per ottenere i *documenti recuperati*: questi, prima di essere propo-

sti all'utente, vengono ordinati (*ranking*) sulla base di un livello di rilevanza determinato da una misura di similarità (figura 1.2).

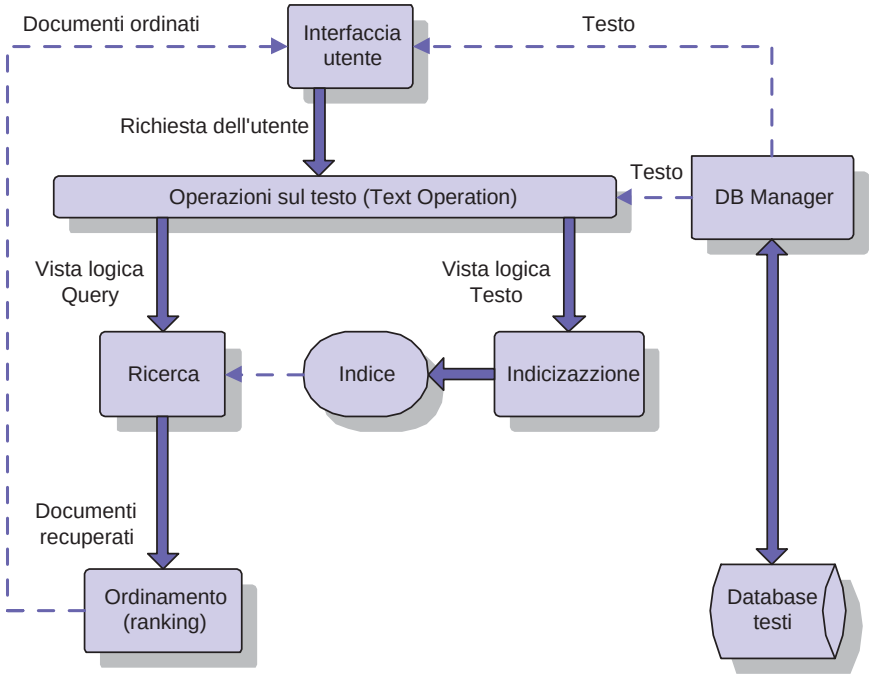


Figura 1.2: Il procedimento dell'Information Retrieval

1.3 I modelli di IR

Come è già stato messo in evidenza, uno dei problemi al centro dei sistemi di IR è quello di predire quali documenti sono rilevanti e quali non lo sono; questa decisione è normalmente dipendente dall'algoritmo di ranking utilizzato, il quale tenta di stabilire, sulla base di una misura di similarità, un semplice ordinamento dei documenti recuperati. I documenti che appaiono in cima a questo ordinamento sono quelli considerati più rilevanti. Pertanto, gli algoritmi di ranking sono al centro dei sistemi di IR; essi operano diversamente a seconda del *modello* di IR adottato.

I tre modelli classici dell'information retrieval sono [3]:

- *Booleano*
- *Vettoriale*

- *Probabilistico*

Nel modello booleano i documenti e le query sono rappresentati come insiemi (*set*) di parole chiave; pertanto diremo che il modello è di tipo *set theoretic*. Nel modello vettoriale, i documenti e le query sono rappresentati come vettori in uno spazio t-dimensionale; pertanto definiremo questo modello *algebrico*. Nel modello probabilistico, il fondamento della modellazione della rappresentazione del documento e della query è la teoria della probabilità; pertanto diremo che il modello è *probabilistico*.

Nel seguito verranno analizzati in particolare i modelli booleano e vettoriale (e relative estensioni), cioè quelli più utilizzati e inerenti alle problematiche affrontate in questo lavoro.

1.3.1 Formalizzazione generale

È possibile formalizzare nel seguente modo un generico modello di IR:

Definizione 1.1 ***Modello di Information Retrieval.*** *Un modello di Information Retrieval è una quadrupla* $[D, Q, F, R(q_i, d_j)]$ *dove*

1. *D è un insieme composto di viste (o rappresentazioni) logiche dei documenti della collezione.*

2. *Q è un insieme di viste (o rappresentazioni) logiche delle richieste di informazioni degli utenti (query).*

3. *F (framework) è l'insieme delle regole alla base della modellazione delle rappresentazioni dei documenti, delle query e delle loro relazioni.*

4. $R(q_i, d_j)$ *è una funzione di ordinamento (ranking) che associa un numero reale a ogni coppia costituita da una query* $q_i \in Q$ *e da una rappresentazione di documento* $d_j \in D$*. Tale ranking definisce un ordine tra i documenti rispetto alla query* q_i*.*

I modelli classici, come si è visto, considerano ogni documento come descritto da un insieme di parole chiave rappresentative dette anche *termini indice*. Sia k_i un termine indice, d_j un documento e $w_{i,j}$ un *peso* associato alla coppia (k_i, d_j). Questo peso quantifica l'importanza del termine indice nel descrivere i contenuti

semantici del documento: maggiore è il peso maggiormente il termine è significativo ed adatto a descrivere e a rappresentare gli argomenti trattati nel testo. Formalizzando:

Definizione 1.2 ***Peso dei termini.*** *Sia t il numero di termini indice nel sistema e k_i un generico termine indice. $K = \{k_1, ..., k_t\}$ è l'insieme di tutti i termini indice. Un peso $w_{i,j} > 0$ è associato a ogni termine indice k_i di un documento d_j. Per un termine indice che non appare nel testo del documento, $w_{i,j}$=0. Al documento d_j è associato un vettore di termini indice $\overrightarrow{d_j}$ rappresentato da $\overrightarrow{d_j} = (w_{1,j}, w_{2,j}, ..., w_{t,j})$. Inoltre, sia g_i una funzione che restituisce il peso associato al termine indice k_i in ogni vettore t-dimensionale (cioè $g_i(\overrightarrow{d_j}) = w_{i,j}$).*

I pesi dei termini indice sono normalmente assunti mutualmente indipendenti.

1.3.2 Il modello booleano

Il modello booleano è un semplice modello basato sulla teoria degli insiemi e sull'algebra booleana. Le query vengono espresse come espressioni booleane con precise semantiche, fornendo a questo modello una notevole semplicità e una chiara formalizzazione. Passando alla caratterizzazione formale:

Definizione 1.3 ***Modello booleano.*** *Per il modello booleano, le variabili peso dei termini indice sono binarie, cioè $w_{i,j} \in \{0, 1\}$. Una query q è una convenzionale espressione booleana. Sia $\overrightarrow{q}_{dnf}$ la forma disgiuntiva normale della query q, $\overrightarrow{q}_{cc}$ ognuna delle componenti congiuntive di $\overrightarrow{q}_{dnf}$. La similarità di un documento d_j alla query q è definita come:*

$$sim(d_j, q) = \begin{cases} 1 & se\ \exists\ \overrightarrow{q}_{cc} \mid (\overrightarrow{q}_{cc} \in \overrightarrow{q}_{dnf}) \bigwedge (\forall k_i :\ g_i(\overrightarrow{d_j}) = g_i(\overrightarrow{q}_{cc})) \\ 0 & altrimenti. \end{cases}$$

Se $sim(d_j, q)$=1 allora il modello booleano predice che il documento d_j è rilevante per la query q. Altrimenti, la predizione è che il documento non è rilevante.

Per meglio chiarire la definizione facciamo questo esempio: formulare la query $q = k_a \bigwedge (k_b \bigvee \neg k_c)$ significa richiedere i documenti che contengono il termine k_a ed il termine k_b, oppure quelli che contengono k_a ma non k_c, e così via. In forma disgiuntiva normale, sulla base della tupla (k_a, k_b, k_c), la query è così espressa:

$\overrightarrow{q}_{dnf} = (1,1,1) \bigvee (1,1,0) \bigvee (1,0,0)$ (figura 1.3). Il documento $\overrightarrow{d_j} = (0,1,0)$, che contiene solo il termine k_b, sarebbe pertanto considerato non significativo.

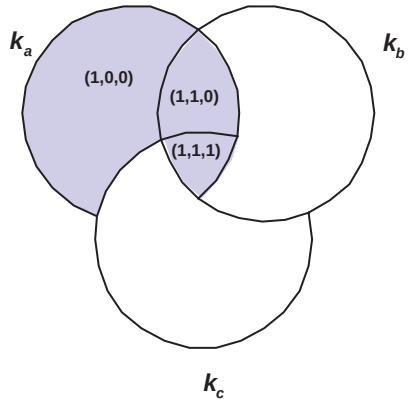


Figura 1.3: Modello booleano: le tre componenti congiuntive della query q

Il modello booleano è un modello semplice e chiaro, che purtroppo presenta alcuni svantaggi: la strategia di recupero dei documenti è basata su un criterio di decisione binario (un documento è rilevante o non lo è), senza nessuna gradazione intermedia. Inoltre, nonostante le espressioni booleane abbiano una precisa semantica, spesso non è semplice esprimere con esse la richiesta di informazione di un utente.

1.3.3 Il modello vettoriale

Il modello vettoriale riconosce che l'utilizzo di pesi binari è troppo limitativo e propone un insieme di regole di base in cui è possibile un match parziale. Per farlo esso assegna ai termini indice dei pesi non-binari, nelle query e nei documenti; questi pesi vengono utilizzati per calcolare il *grado di somiglianza* tra ogni documento memorizzato nel sistema e la query dell'utente.

Definizione 1.4 ***Modello vettoriale.*** *Nel modello vettoriale, il peso $w_{i,j}$ associato alla coppia (k_i, d_j) è positivo e non-binario. Inoltre, anche ai termini indice della query viene assegnato un peso. Sia $w_{i,q}$ il peso associato alla coppia $[k_i, q]$, dove $w_{i,q} \geq 0$. Il vettore della query $\overrightarrow{q}$ è quindi definito come $\overrightarrow{q} = (w_{1,q}, w_{2,q}, ..., w_{t,q})$, dove t è il numero totale dei termini indice nel sistema. Come sempre, il vettore di un documento d_j è rappresentato da $\overrightarrow{d_j} = (w_{1,j}, w_{2,j}, ..., w_{t,j})$.*

Il modello vettoriale si propone di valutare il grado di similarità del documento d_j rispetto alla query q come la correlazione tra i vettori $\overrightarrow{d_j}$ e $\overrightarrow{q}$. Questa correlazione può essere quantificata, ad esempio, come il *coseno dell'angolo* tra i due vettori (figura 1.4).

Pertanto:

$$sim(d_j, q) = \frac{\overrightarrow{d_j} \cdot \overrightarrow{q}}{|\overrightarrow{d_j}||\overrightarrow{q}|} = \frac{\sum_{i=1}^{t} w_{1,j}\, w_{1,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2}\ \sqrt{\sum_{j=1}^{t} w_{i,q}^2}}$$

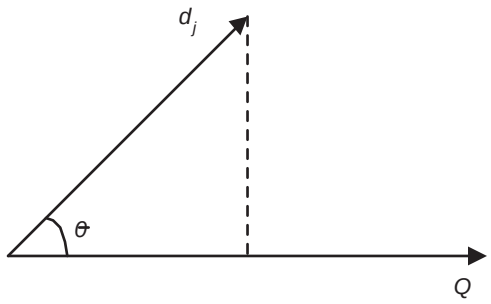


Figura 1.4: Modello vettoriale: il coseno di θ adottato come $sim(d_j, q)$

I pesi dei termini possono essere calcolati in molti modi. L'idea alla base delle tecniche più efficienti è legata al concetto di *cluster*, proposto da Salton [35].

Si pensi ai documenti come a una collezione C di oggetti e alla query dell'utente come a una (vaga) specificazione di un insieme A di oggetti. Il problema dell'IR può essere allora ridotto a quello di determinare quali documenti sono nel set A e quali no; in particolare in un problema di clustering occorre risolvere due sotto-problemi. Occorre determinare quali sono le caratteristiche che descrivono al meglio gli oggetti dell'insieme A (si parla allora di similarità *intra-cluster*) e quali sono le caratteristiche che meglio distinguono gli oggetti dell'insieme A dagli altri oggetti della collezione C (dissimilarità *inter-cluster*).

Nel modello vettoriale, la similarità intra-cluster è quantificata misurando la frequenza di un termine k_i in un documento d_j: questa viene detta *fattore tf* (*term frequency*) e misura quanto un termine descrive i contenuti del documento. La dissimilarità inter-cluster è invece quantificata misurando l'inverso della

frequenza di un termine k_i tra i documenti della collezione (*fattore idf* o *inverse document frequency*).

Formalizzando questi concetti:

Definizione 1.5 ***Pesi dei termini (modello vettoriale).*** *Sia N il numero totale di documenti del sistema e n_i il numero dei documenti i cui compare il termine indice k_i. Sia $freq_{i,j}$ la frequenza del termine k_i nel documento d_j (cioè il numero di volte in cui il termine k_i è usato nel testo del documento d_j). Allora, la frequenza normalizzata $f_{i,j}$ del termine k_i nel documento d_j è data da*

$$f_{i,j} = \frac{freq_{i,j}}{max_l freq_{l,j}}$$

dove il massimo è calcolato tra tutti i termini menzionati nel testo del documento d_j. Se il termine k_i non compare nel documento d_j allora $f_{i,j} = 0$. Inoltre, sia idf_i, la inverse document frequency di k_i, così definita:

$$idf_i = log\frac{N}{n_i}$$

I più noti schemi di peso dei termini usano pesi forniti da

$$w_{i,j} = f_{i,j} \times idf_i$$

o da una variante di questa formula. Questi vengono detti schemi tf-idf.

I vantaggi del modello vettoriale sono diversi, in particolare l'utilizzo di pesi migliora la performance del recupero dei documenti, consente il recupero di documenti che *approssimano* le condizioni di query. Inoltre, la formula di ranking con il coseno ordina i documenti rispetto al grado di somiglianza con la query. Un difetto può essere invece riconosciuto nel fatto che i termini indice sono sempre assunti mutualmente indipendenti.

1.3.4 Modello vettoriale generalizzato

Come si è visto, il modello vettoriale classico assume l'indipendenza dei termini indice. Questa assunzione è interpretabile come segue:

Definizione 1.6 ***Indipendenza dei termini indice.*** *Sia $\overrightarrow{k_i}$ un vettore associato al termine indice k_i. L'indipendenza dei termini indice implica che l'insieme dei vettori $\{\overrightarrow{k_1}, \overrightarrow{k_2}, ..., \overrightarrow{k_t}\}$ è linearmente indipendente e forma una base per il sottospazio di interesse. La dimensione di questo spazio è il numero t di termini indice nella collezione.*

Spesso, l'indipendenza tra i termini indice è interpretata più ristrettivamente, sottintendendo ortonormalità tra i vettori dei termini, cioè per ogni coppia di vettori $\overrightarrow{k_i}$ e $\overrightarrow{k_j}$ si ha $\overrightarrow{k_i} \cdot \overrightarrow{k_j} = 0$. Il *modello vettoriale generalizzato* [50] si basa invece sull'assunzione di non ortonormalità: in questo modello, due vettori dei termini possono essere non ortogonali.

Definizione 1.7 ***Mintermini di co-occorrenza.*** *Dato l'insieme* $\{\overrightarrow{k_1}, \overrightarrow{k_2}, ..., \overrightarrow{k_t}\}$ *di termini indice di una collezione, sia* $w_{i,j}$ *il peso associato alla coppia termine-documento* $[k_i, d_j]$. *Se i pesi* $w_{i,j}$ *sono tutti binari allora tutti i possibili pattern di co-occorrenza dei termini (nei documenti) possono essere rappresentati da un insieme di* 2^t *mintermini dati da* $m_1 = (0, 0, ...0)$, $m_2 = (1, 0, ..., 0)$, ..., $m_{2^t} = (1, 1, ..., 1)$. *Sia* $g_i(m_j)$ *una funzione che restituisce il peso* $\{0,1\}$ *del termine indice* k_i *nel mintermine* m_j.

In questo modo il mintermine m_1 (per cui $g_i(m_1) = 0$), ad esempio, punta a tutti i documenti che non contengono nessuno dei termini indice. L'idea di fondo è quella di introdurre un set di vettori ortogonali a coppie $\overrightarrow{m_i}$ associati all'insieme dei mintermini e adottare come base questo insieme di vettori.

Definizione 1.8 *Definiamo il seguente insieme di vettori* $\overrightarrow{m_i}$

$$\begin{aligned} \overrightarrow{m_1} &= (1, 0, ..., 0, 0) \\ \overrightarrow{m_2} &= (0, 1, ..., 0, 0) \\ &... \\ \overrightarrow{m_{2^t}} &= (0, 0, ..., 0, 1) \end{aligned}$$

dove ogni vettore $\overrightarrow{m_i}$ *è associato al rispettivo mintermine* m_i.

Per determinare il vettore dei termini indice $\overrightarrow{k_i}$ associato al termine indice k_i è allora sufficiente sommare i vettori di tutti i mintermini m_r in cui il termine k_i è nello stato 1 e normalizzare. Pertanto,

$$\overrightarrow{k_i} = \frac{\sum_{\forall r, g_i(m_r)=1} c_{i,r} \overrightarrow{m_r}}{\sqrt{\sum_{\forall r, g_i(m_r)=1} c_{i,r}^2}}$$

$$c_{i,r} = \sum_{d_j | g_l(d_j) = g_l(m_r)\ \forall l} w_{i,j}$$

Per ogni vettore $\overrightarrow{m_r}$, è definito un *fattore di correlazione* $c_{i,r}$ che somma i pesi $w_{i,j}$ associati ai termini indice k_i e ad ogni documento d_j i cui pattern di occorrenza dei termini coincidono esattamente con quello del mintermine m_r. Così, un mintermine è di interesse solo se almeno uno dei documenti della collezione condivide il suo pattern.

Nel modello vettoriale classico, un documento d_j e una query q erano espressi da $\overrightarrow{d_j} = \sum_{\forall i} w_{i,j}\overrightarrow{k_i}$ e $\overrightarrow{q} = \sum_{\forall i} w_{i,q}\overrightarrow{k_i}$ rispettivamente. Nel modello vettoriale generalizzato, queste rappresentazioni possono essere tradotte nello spazio dei mintermini mediante le equazioni viste, e i vettori $\overrightarrow{d_j}$ e $\overrightarrow{q}$ sono quindi utilizzati per calcolare il ranking di ordinamento. Tale ranking combina i pesi standard $w_{i,j}$ con i fattori di correlazione $c_{i,r}$, fornendo, rispetto al modello classico, una solitamente migliore accuratezza nella ricerca a scapito di una maggiore complessità computazionale.

1.4 Valutazione dei risultati di un sistema di IR

In un sistema di Data Retrieval, le metriche di maggiore interesse e quelle normalmente utilizzate per valutare il sistema, sono il tempo di risposta e lo spazio richiesto. In questo caso, si controllano la performance delle strutture di indice, l'interazione con il sistema operativo, i ritardi nei canali di comunicazione, gli overhead introdotti dai molti strati software presenti: in una parola ciò che interessa è l'*efficienza*.

In un sistema di Information Retrieval altre metriche sono di interesse oltre al tempo e allo spazio. I sistemi di IR richiedono di valutare anche quanto è preciso l'insieme di risposta a una query: si parla in questo caso di *efficacia* del sistema di IR.

1.4.1 Una misura dell'efficacia: Richiamo e Precisione

Si consideri una richiesta di informazioni I ed il suo insieme R di documenti rilevanti. Sia $|R|$ il numero di documenti in questo insieme. Si assuma che una data strategia di recupero (che si sta valutando) elabori la richiesta di informazioni I e generi un insieme di documenti di risposta A. Sia $|A|$ il numero di documenti di questo insieme. Inoltre, sia $|Ra|$ il numero dei documenti nell'intersezione degli insiemi R ed A (figura 1.5).

Definizione 1.9 ***Richiamo (Recall).*** *Si definisce Richiamo o Recall la frazione di documenti rilevanti (insieme R) che è stata recuperata, cioè*

$$Richiamo = \frac{|Ra|}{|R|}$$

Definizione 1.10 ***Precisione.*** *Si definisce Precisione la frazione di documenti recuperati (insieme A) che è rilevante, cioè*

$$Precisione = \frac{|Ra|}{|A|}$$

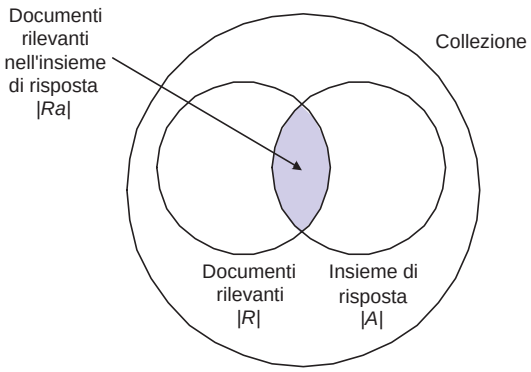


Figura 1.5: Precisione e richiamo per una data richiesta di informazione

1.4.2 Curva precisione-richiamo

Richiamo e precisione assumono che siano stati esaminati tutti i documenti nell'insieme di risposta A. Tuttavia, all'utente non viene normalmente mostrato tutto l'insieme dei documenti in una sola volta. I documenti in A vengono prima ordinati in base al grado di rilevanza, quindi l'utente esamina la lista ordinata cominciando dall'alto. In questa situazione, le misure di precisione e richiamo cambiano man mano che questi procede con l'esame dell'insieme delle risposte A. Pertanto, per una corretta valutazione è necessario disegnare una *curva precisione-richiamo* [3].

Si procede nel modo seguente: immaginiamo che l'insieme R_q contenga i documenti rilevanti alla query q, ad esempio

$R_q = \{d_3, d_5, d_9, d_{25}, d_{39}, d_{44}, d_{56}, d_{71}, d_{89}, d_{123}\}$

Ipotizziamo che l'algoritmo da valutare presenti il seguente elenco ordinato di documenti:

1. $\Rightarrow d_{123}$	6. $\Rightarrow d_9$	11. d_{38}
2. d_{84}	7. d_{511}	12. d_{48}
3. $\Rightarrow d_{56}$	8. d_{129}	13. d_{250}
4. d_6	9. d_{187}	14. d_{113}
5. d_8	10. $\Rightarrow d_{25}$	15. $\Rightarrow d_3$

I documenti rilevanti alla query q sono evidenziati da una freccia ($\Rightarrow$). Si ricavano queste considerazioni: il documento d_{123} ha il massimo ranking ed è rilevante. Inoltre, il documento corrisponde al 10% di tutti i documenti rilevanti dell'insieme R_q. Pertanto avremo una precisione del 100% ad un richiamo del 10%. Poi, il documento d_{56} che ha un ranking di 3 è il prossimo documento rilevante: a questo punto la precisione sarà di circa il 66% con il 20% di richiamo. Proseguendo in questo modo si può ottenere il grafico mostrato in figura 1.6.

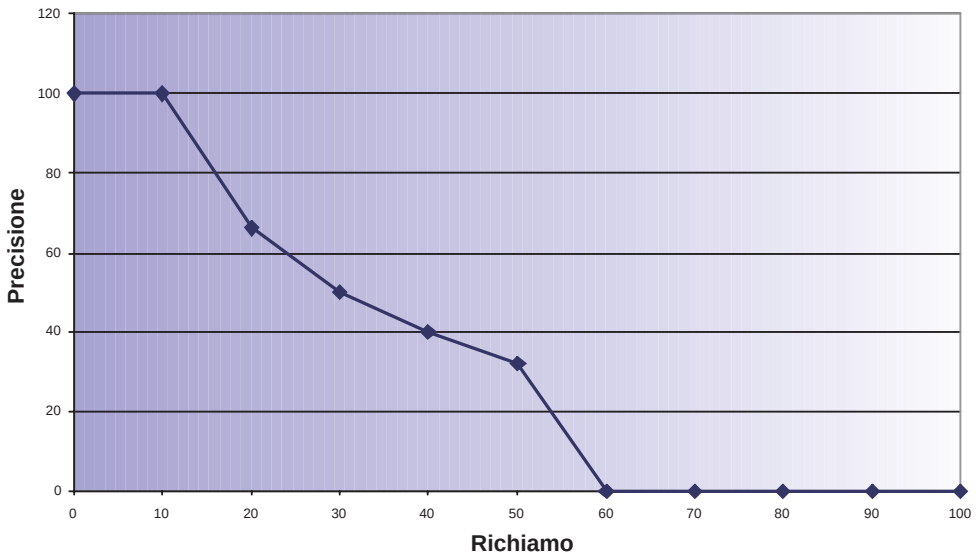


Figura 1.6: Curva Richiamo-Precisione: un esempio

Infine, per valutare un algoritmo di IR su tutte le query di test, si mediano i valori di precisione a ogni livello di richiamo:

$$\overline{P}(r) = \sum_{i=1}^{N_q} \frac{P_i(r)}{N_q}$$

dove $\overline{P}(r)$ è la precisione media al livello di richiamo r, N_q è il numero delle query e $P_i(r)$ la precisione al livello di richiamo r per la i-esima query.

1.4.3 Valutazione con singoli valori

È possibile ottenere delle valutazioni complessive di un sistema di IR anche utilizzando un solo valore. Due sono le tecniche più utilizzate:

- *Precisione media ai documenti rilevanti*: si fa la media dei valori di precisione ottenuti dopo che viene osservato (nel ranking) ciascun nuovo documento rilevante. Nell'esempio precedente, i valori di precisione osservati in seguito a ciascun documento rilevante sono 1, 0.66, 0.5, 0.4, e 0.3. Così, la *precisione media ai documenti rilevanti* è data da (1+0.66+0.5+0.4+0.3)/5 o 0.57. Questa metrica favorisce i sistemi che recuperano velocemente i documenti rilevanti.

- *Precisione R*: si calcola la precisione alla R-esima posizione nel ranking, dove R è il numero totale di documenti rilevanti per la query corrente (cioè il numero di documenti del set R_q). Nell'esempio, il valore della precisione R è 0.4, poiché $R = 10$ e ci sono quattro documenti rilevanti tra i primi dieci del ranking.

Si tratta di parametri significativi, anche se bisogna tenere conto che utilizzare un singolo valore per riassumere l'intero comportamento di un sistema di IR sulle varie query può divenire piuttosto impreciso.

1.5 Le query di un sistema IR

Una query rivolta ad un sistema di IR può essere di vari tipi; di seguito vengono spiegati i più comuni.

- *Query basate su parole chiave*: possono essere molto semplici, come query di *singole parole*, o più complesse, come query di più parole in un *contesto*. Ad esempio può essere richiesta la ricerca di parole contenute in una *frase*,

o che siano semplicemente *prossime*. Sempre in questo campo esistono inoltre query *booleane*, che utilizzano cioè gli operatori *OR*, *AND*, e così via, fino ad arrivare a query in *linguaggio naturale*.

- *Query strutturali*: query che riguardano la struttura del testo, che può essere ad esempio basato su ipertesti o gerarchie.

- *Query basate su pattern matching*: un *pattern* è un insieme di caratteristiche sintattiche che devono ricorrere in un segmento di testo. I pattern più usati sono: parole, prefissi, suffissi e sottostringhe; è inoltre possibile *tollerare errori*, specificando non solo la parola ma anche la soglia di errore. Nonostante ci siano molti modelli per la similarità tra le parole, il più accettato nell'IR di testi è la *Levenshtein distance* o *edit distance*.

L'*edit distance* tra due stringhe è il minimo numero di inserimenti, cancellazioni e sostituzioni di caratteri necessari a rendere le due stringhe uguali.

Come si vedrà nel Capitolo 3 della seconda parte (cui si rimanda per una definizione più precisa e formale), proprio il concetto di edit distance è stato approfondito ed esteso nel progetto EXTRA, dove rappresenta il vero cardine della ricerca di similarità tra frasi.

1.6 Operazioni sul testo

Le *Text Operation* [3] (operazioni sul testo) riguardano la pre-elaborazione del testo di un documento. Questa procedura può essere divisa in più parti:

1. analisi lessicale del testo, con trattamento di cifre, punteggiatura, ecc;

2. eliminazione delle parole troppo frequenti, le cosiddette *stopword*, inutili per discriminare un documento (ad esempio gli articoli);

3. *stemming* (o normalizzazione) delle parole rimanenti.

In questa sede verrà brevemente descritta proprio l'operazione di stemming, che come si vedrà è piuttosto importante nel lavoro proposto.

1.6.1 Stemming

Di solito, l'utente specifica una parola nella query ma solo una variante di questa parola è presente nel relativo documento. I plurali, le coniugazioni sono esempi di variazioni sintattiche che impediscono una perfetta corrispondenza tra una parola di query e la rispettiva parola del documento. Questo problema può essere risolto con la sostituzione delle parole con i rispettivi *stem*. Uno *stem* è la porzione della parola che rimane dopo la rimozione dei suoi affissi (prefissi e suffissi). Un tipico esempio di stem, in inglese, è la parola *connect*, che è lo stem delle varianti *connected, connecting, connection*, e *connections*.

$$\left.\begin{array}{l} connected \\ connecting \\ connection \\ connections \end{array}\right\} \rightarrow connect$$

Lo stemming è pertanto utile per migliorare la performance di un sistema di IR poiché riduce le varianti di una stessa parola-radice ad un concetto comune. Inoltre, lo stemming ha l'effetto secondario di ridurre la dimensione delle strutture di indice poiché è ridotto il numero di termini indice distinti.

1.7 Indicizzazione e Inverted index

Dopo aver elaborato (e ridotto) l'elenco delle parole, è necessario indicizzarle. La forma di indice più diffusa nell'ambito di IR è l'*inverted index* [3]. Un inverted index è un meccanismo orientato alle parole per indicizzare una collezione di testo, al fine di velocizzare l'operazione di ricerca. La struttura dell'inverted index è composta da due elementi (figura 1.7): il *vocabolario* e le *occorrenze*.

Il *vocabolario* è l'insieme di tutte le parole del testo; per ognuna di queste è conservata una lista di tutte le posizioni nel testo dove essa compare. L'insieme di tutte queste liste viene chiamato *occorrenze*.

Le posizioni possono fare riferimento a caratteri o a parole. Le posizioni relative alle parole semplificano le query di prossimità o di frase, mentre le posizioni relative ai caratteri facilitano l'accesso diretto alle posizioni corrispondenti nel testo.

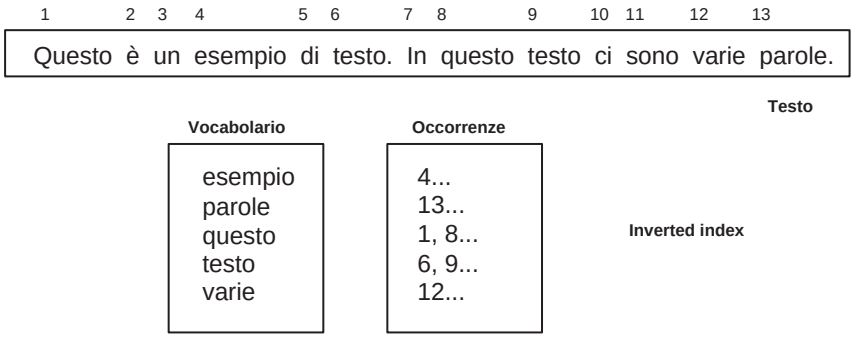


Figura 1.7: Inverted Index: un esempio

1.8 IR e Ricerca di Similarità: Stato dell'arte

In questa sezione viene fatta una panoramica sullo stato della ricerca scientifica in ambito di Information Retrieval. In questo caso non siamo però interessati alle problematiche classiche, quelle che come si è visto sono tipiche di questa scienza (ricerca di documenti attinenti a una query), ma a una sua applicazione ben specifica: quella della *ricerca di similarità tra frasi*, alla base del nostro progetto.

1.8.1 La Ricerca scientifica

Il problema della ricerca di similarità tra frasi è un problema del tutto nuovo in campo scientifico: date due frasi, si tratta di stabilire *se* e *quanto* sono vicine. Andando ancora oltre: data una frase e un insieme di frasi con cui confrontarla, occorre recuperare le frasi che più si avvicinano a quella data. Un problema che nessun ricercatore ha mai studiato specificamente; tuttavia, proprio in ambito di Information Retrieval si sono svolti parecchi studi che, pur non riguardanti specificamente questo problema, si sono rivelati comunque molto utili per affrontarlo al meglio nel nostro progetto.

Pattern matching approssimato

Questo è uno dei problemi classici in ambito di IR (presentato nella sezione 1.5) che più hanno in comune con quello della similarità. Si tratta anche in questo caso di ricercare quanto un dato pattern è *simile* ad un altro e, se la somiglianza rientra in una certa soglia, di proporlo nel risultato della ricerca.

In particolare, ad esempio, Gravano, Ipeirotis, Jagadish, Koudas, Muthukrishnan e Srivastava [14] propongono un interessante algoritmo per il join approssimato di stringhe presenti in un database: le stringhe cioè faranno join non solo se *esattamente* uguali, ma anche se uguali solo in modo *approssimato*. La finalità è molto diversa dalla nostra: essi intendono ovviare con questo metodo a possibili errori (ad esempio di battitura) che, normalmente, esistono nei dati. Per esempio, i due diversi inserimenti "Smith" e "Smiht", quest'ultimo errato, verrebbero comunque riconosciuti e posti in join dal DBMS.

Le tecniche che stanno alla base, tuttavia, sono molto potenti e flessibili e si riveleranno utili anche per il nostro problema: il motore di similarità di EXTRA prende spunto proprio dai concetti di q-grammi e di edit distance proposti in questo lavoro (si rimanda alla seconda parte per una spiegazione dettagliata).

Proximity Search

Un altro problema classico dell'IR e un'altra notevole fonte di ispirazione è senza dubbio quello del *Proximity Search*. Anche in questo caso, le finalità dei ricercatori sono diverse dalle nostre; in particolare ci si occupa di effettuare una ricerca in documenti di una serie di parole che devono comparire *vicine* le une alle altre, se non addirittura *in sequenza*.

Un interessante studio a questo riguardo è stato svolto da Grossman, Holmes, Frieder e Roberts [15], in cui viene proposto un metodo per un'efficace implementazione SQL di questo tipo di query.

Spostando il problema dai *documenti* alle *frasi*, si capisce che anche questo problema ha molto a che fare con quello della similarità tra frasi: una frase è simile ad un altra se presenta le parole fondamentali approssimativamente nello stesso ordine.

Natural Language Retrieval

Molti studi sono stati svolti anche nel campo del *Natural Language Processing (NLP)* applicato all'Information Retrieval. Queste applicazioni si differenziano dal NLP classico in quanto di prefiggono uno scopo diverso. Come si sa, il problema tipico dell'IR è capire se un documento è rilevante o no rispetto a una data query: per ottenere questo scopo non è necessaria la piena comprensione

e interpretazione del documento, cosa invece cui mira la ricerca in puro ambito NLP.

Croft e Lewis [8] propongono un linguaggio chiamato *REST (Representation for Science and Technology)*: si tratta di un linguaggio basato su *frame* che permette di rappresentare il contenuto di un documento in termini di un insieme di concetti scientifici di base. Dopo aver indicizzato questi contenuti per i documenti del DB, nello stesso modo si crea una rappresentazione REST della query e si esegue la ricerca. A titolo di esempio, in figura 1.8 viene mostrata la rappresentazione REST della query "Probabilistic analyses of Quicksort, a divide and conquer algorithm".

```
(STUDY-1
   APPEARANCE; analysis)
(STUDY-2
   IS-A(DEF): STUDY-1
   ARGUMENT-OF(DEF): RELATIONSHIP-1)
(STUDY-3
   IS-A(DEF): STUDY-2
   INTEREST(DEF): METHOD-3)
(RELATIONSHIP-1
   APP: probabilistic)
(METHOD-1
   APP: algorithm)
(METHOD-2
   IS-A(DEF): METHOD-1
   USES(DEF): ACTION-1)
(METHOD-3
   IS-A(DEF): METHOD-2
   APP: Quicksort)
(ACTION-1
   APP: divide and conquer)
```

Figura 1.8: NLP: un esempio di rappresentazione REST

Altri lavori sono quelli di Strzalkowski e Carballo [40] (un parser in grado di riconoscere la struttura delle frasi e le parole più significative) e di Arampatzis, Tsoris, Koster e Van der Weide [2], che propongono un sistema in grado di agire a più livelli su lessico, sintassi, morfologia e semantica.

In generale si tratta di sistemi che ottengono sì buoni risultati ma che possono essere applicati solo ad ambiti molto ristretti (una lingua e un settore ben precisi), data l'enorme quantità di informazioni di base che richiedono per poter funzionare.

1.8.2 I Sistemi commerciali

Ovviamente non esistono strumenti commerciali specificamente pensati per la ricerca di similarità tra frasi; come per la ricerca scientifica, tuttavia, alcune proposte offrono interessanti spunti o risultano in qualche modo attinenti a questa problematica. Esse vengono pertanto brevemente descritte nei paragrafi seguenti e riguardano principalmente:

- Sistemi per Text Retrieval su Web
- Estensioni ai DBMS per Text Retrieval

Sistemi per Text Retrieval su Web

In ambito commerciale, sono disponibili su Internet diversi sistemi con lo scopo di catalogare, organizzare e ricercare documenti di vario genere. In particolare, è possibile accedere a diversi sistemi che permettono la ricerca in *Digital Library* di letteratura scientifica; uno dei più significativi a questo proposito è *CiteSeer* [20], nato da un progetto al NEC Research Institute.

CiteSeer è un sistema per la creazione di Digital Library di argomento scientifico, operazione che include una efficiente identificazione degli articoli, la loro indicizzazione, la creazione automatica di indici di citazioni, l'estrazione di informazioni, la visualizzazione di riassunti in relazione alle query, e altro ancora.

Ciò che ha attratto la nostra attenzione, e che fa rientrare questi sistemi nell'interesse di questa panoramica, è la possibilità di proporre, dato un documento, tutti quei documenti che risultano in qualche modo *simili* ad esso. Addirittura, CiteSeer vanta la possibilità di proporre i documenti che siano *simili a livello delle frasi*: qualcosa che, a prima vista, sembra esattamente quello che ci interessa.

In realtà, un'analisi più approfondita del funzionamento di questi sistemi e di CiteSeer in particolare fa capire come nemmeno in questo ambito si risolva in modo soddisfacente il problema della similarità. Quello che viene fatto è semplicemente mantenere un database con tutte le frasi dei vari documenti e accedere a questo per calcolare la percentuale di frasi *identiche*: le coppie di documenti con un'alta percentuale vengono considerate simili.

Ai nostri fini è ovviamente un approccio molto limitativo, ma fornisce comunque alcune idee interessanti, come ad esempio l'utilizzo di una percentuale di somiglianza.

Estensioni ai DBMS per Text Retrieval

Esistono inoltre diversi pacchetti che si propongono di estendere le funzionalità dei diversi DBMS commerciali all'ambito dell'IR. In particolare, verrà qui ricordato *Oracle interMedia Text* per il DBMS *Oracle*: esso permette l'indicizzazione di documenti di testo e l'esecuzione di interrogazioni tipiche di IR (come, ad esempio, la ricerca di documenti che contengono una certa parola), utilizzando comuni comandi SQL. In altre parole, esso estende al DBMS Oracle tutte le operazioni svolte tipicamente da un Information Retrieval System.

Basandosi sull'utilizzo di questi strumenti è possibile costruire un algoritmo per la ricerca di similarità tra frasi: è quello che è stato fatto, ad esempio, da F. Gavioli nella sua Tesi di Laurea [13]. Viene qui brevemente descritto l'approccio da questi adottato, cui sarà interessante confrontare la relativa parte del nostro progetto; i passi fondamentali sono i seguenti:

1. Stemming (vedere la sezione 1.6.1 per una spiegazione) ed inserimento nel DB delle frasi di riferimento

2. Indicizzazione mediante interMedia di tali frasi

3. Stemming della frase di query (da ricercare)

4. Generazione dei relativi *q-grammi* (cioè sequenze di q parole consecutive)

5. Estrazione delle frasi che presentano il maggior numero di q-grammi in comune, mediante un'istruzione SQL di questo tipo:

   SELECT <codice>, COUNT(∗)
   FROM (**SELECT** <codice>
   FROM <tabella frasi>
   WHERE CONTAINS (<colonna frase>, '<trigramma>') > 0
   UNION **ALL**
   SELECT <codice> ...
   −− una istruzione SELECT per ogni trigramma
)

   dove *CONTAINS* è un'estensione interMedia per la ricerca di frasi all'interno di testi. Il risultato è l'insieme dei testi che contengono esattamente

la frase in input. Il valore di similarità prodotto da *CONTAINS* è determinato dal prodotto dei pesi dei termini che compaiono nella frase in input con i pesi dei termini dei testi risultato della query.

In particolare, viene scelto di utilizzare trigrammi (q=3) e di utilizzare la seguente metrica per quantificare la somiglianza tra la coppia di frasi: il rapporto tra il numero di trigrammi comuni e il numero di trigrammi che compongono la frase da tradurre.

Come già ricordato, nella seconda parte di questo documento (Capitolo 6) verranno confrontati, sia in efficienza che in efficacia, i risultati conseguiti dal modulo di ricerca di similarità del nostro sistema, EXTRA, con quelli ottenibili da altri approcci, tra cui appunto quello appena descritto.

Nel prossimo capitolo, invece, si concentrerà l'attenzione su un altro importante argomento, anch'esso centrale ed indispensabile per collocare il progetto EXTRA nel giusto contesto: quello della traduzione assistita, dalla teoria di base ai principali strumenti disponibili.

Capitolo 2

Analisi della traduzione assistita

2.1 Introduzione alla Machine Translation

Il termine *Machine Translation* (MT) [16] indica genericamente la traduzione mediante computer. Nel suo significato più ampio, MT include in realtà applicazioni per computer come compilatori, programmi di compressione, ecc., che convertono un file da un linguaggio per computer ad un altro. Tuttavia, quello che veramente ci interessa in questo ambito è il cosiddetto *Natural Language Processing (NLP)*, cioè l'elaborazione del linguaggio naturale.

Ci sono quattro principali tipologie di traduzione, che vedono il ruolo della macchina via via sempre più fondamentale:

- *Traduzione umana.* Un traduttore umano esegue tutti i passi del processo di traduzione, utilizzando il computer solo come word processor.

- *Traduzione umana assistita dalla macchina*, conosciuta come *Machine-Assisted Human Translation (MAHT).* La traduzione è eseguita da un traduttore umano, ma questi usa il computer come uno strumento per migliorare o velocizzare il processo di traduzione. Questa è anche detta traduzione assistita dal computer (*Computer-Aided Translation* - CAT).

- *Traduzione automatica assistita dall'uomo*, conosciuta anche come *Human-Assisted Machine Translation (HAMT).* Il testo nel linguaggio sorgente è modificato da un traduttore umano prima, durante o dopo essere stato tradotto dal computer.

- *Traduzione completamente automatica*, conosciuta come *Fully Automatic Machine Translation (FAMT)*. Il testo nel linguaggio sorgente è fornito al computer sotto forma di file e questo produce automaticamente una traduzione, senza nessun intervento umano.

Ovviamente la traduzione umana non verrà ulteriormente analizzata; è invece interessante approfondire le tecniche e le assunzioni che stanno alla base delle rimanenti tre categorie.

2.2 I paradigmi dei sistemi di MT

In questa sezione vengono mostrati e discussi le più recenti classi di paradigmi di Machine Translation, quelle che i ricercatori stanno investigando in questi ultimi anni. I paradigmi di ricerca possono essere suddivisi in due grandi categorie [10]:

1. paradigmi fondati su tecniche linguistiche;
2. paradigmi che non usano nessuna tecnica linguistica.

La separazione tra approcci *linguistic-based* e *non-linguistic-based* illustra una dicotomia divenuta evidente sopratutto di recente. Nel seguito verranno mostrate le idee alla base di entrambe le tipologie, evidenziandone pregi e difetti.

2.2.1 Paradigmi linguistic-based

Fino a poco fa, la maggior parte dei ricercatori si sono occupati di Machine Translation con fondamenti linguistici. Questi sistemi si sforzano di utilizzare i vincoli della sintassi, del lessico e della semantica per comprendere il linguaggio sorgente e produrre, nel lignuaggio destinazione, una resa appropriata della frase da tradurre (figura 2.1).

Brevemente, vengono qui esaminati i principali tipi di paradigmi rientranti in questa categoria [10]:

- MT basata su vincoli (*Constraint-Based MT - CBMT*)
- MT basata su conoscenza (*Knowledge-Based MT - KBMT*)
- MT basata su lessico (*Lexical-Based MT - LBMT*)
- MT basata su regole (*Rule-Based MT - RBMT*)

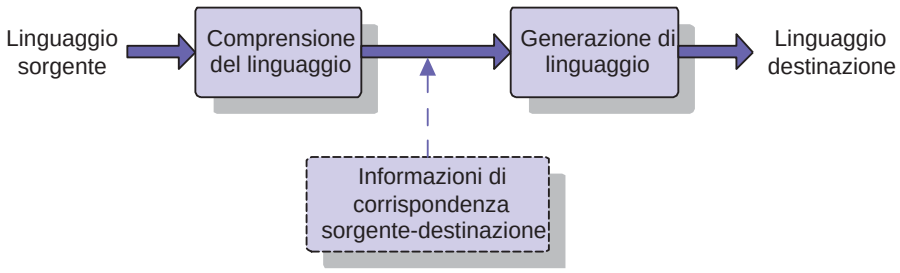


Figura 2.1: Funzionamento dei paradigmi linguistic-based

Constraint-Based MT

Alla base di questa tecnica c'è la cosiddetta *struttura funzionale*, o *struttura-f*: si tratta di una struttura gerarchica che evidenzia il ruolo dei vari componenti della frase (soggetto, verbo, ecc.) e la loro relazione. In questo modo la traduzione è basata su equazioni che mettono in corrispondenza le strutture-f del linguaggio sorgente con quelle del linguaggio di destinazione. A titolo di esempio, in figura 2.2 è mostrata la struttura-f per questa coppia di frasi in inglese e francese:

E: The baby just fell

F: Le bébé vient de tomber

$$\begin{bmatrix} PRED & `JUST((\uparrow ARG))' \\ ARG & \begin{bmatrix} PRED & `FALL((\uparrow SUBJ))' \\ TENSE & PAST \\ SUBJ & \begin{bmatrix} PRED & `BABY' \\ NUM & SG \\ SPEC & \begin{bmatrix} DEF & + \\ PRED & `THE' \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} PRED & `VENIR((\uparrow SUBJ)(\uparrow XCOMP))' \\ SUBJ & \begin{bmatrix} PRED & `BÉBÉ' \\ GENDER & MASC \\ NUMB & SG \\ SPEC & \begin{bmatrix} DEF & + \\ PRED & `LE' \end{bmatrix} \end{bmatrix} \\ XCOMP & \begin{bmatrix} PRED & `TOMBER((\uparrow SUBJ))' \\ COMPL & DE \\ TENSE & INF \\ SUBJ & [BÉBÉ] \end{bmatrix} \end{bmatrix}$$

Figura 2.2: CBMT: la struttura-f delle frasi di esempio

Knowledge-Based MT

Questo paradigma si concentra sullo sviluppo di informazione lessicale incredibilmente dettagliata di tipo morfologico, sintattico, ma anche semantico: a ogni frase viene associata una rappresentazione in testo-interlingua (ILT) che evidenzia non solo la natura degli elementi chiave ma anche il loro presunto significato, sulla base del contesto e del modo cui si relazionano alle parole circostanti.

Lexical-Based MT

Questi sistemi si basano sulla costruzione di grammatiche ad albero che vengono collegate da un linguaggio all'altro mediante un lessico bilingue; questo associa direttamente gli alberi sorgente e destinazione attraverso collegamenti tra gli oggetti lessicali. In pratica, questo approccio utilizza elementi di informazione bilingui, ognuno contenente una corrispondenza tra frase sorgente e destinazione. Uno svantaggio di questo approccio è che richiede il mantenimento di *interi* alberi per ogni coppia tradotta.

Rule-Based MT

Questi sistemi si fondano sull'utilizzo di diversi livelli di regole linguistiche. Le cosiddette *regole-M* sono regole che preservano il significato, e collegano gli alberi sintattici alle corrispondenti strutture di significato; le *regole-S*, invece, si disinteressano della semantica e mettono in corrispondenza elementi lessicali e alberi sintattici.

Non vengono forniti ulteriori esempi per non appesantire la trattazione di troppi dettagli.

In conclusione, si può dire che tutti questi paradigmi sono in grado di fornire traduzioni di discreta qualità e completamente automatiche. Hanno però una grandissima limitazione: possono essere applicati soltanto ad ambiti ben ristretti e definiti, poiché la quantità di informazione richiesta sarebbe altrimenti proibitiva, sia come costo che come tempi per produrla e per gestirla.

2.2.2 Paradigmi non-linguistic-based

Negli ultimi anni i ricercatori hanno investigato soprattutto paradigmi di Machine Translation che non fossero basati su teorie linguistiche o su specifiche

proprietà del linguaggio. Questa ricerca è stata resa possibile dai rapidi avanzamenti nella potenza computazionale e nella crescente disponibilità di dizionari leggibili dalla macchina e di corpus di testi monolingui e bilingui. Questi approcci dipendono infatti dall'esistenza di grandi corpus, utilizzati come dati "di addestramento" o come database di traduzioni esistenti (figura 2.3).

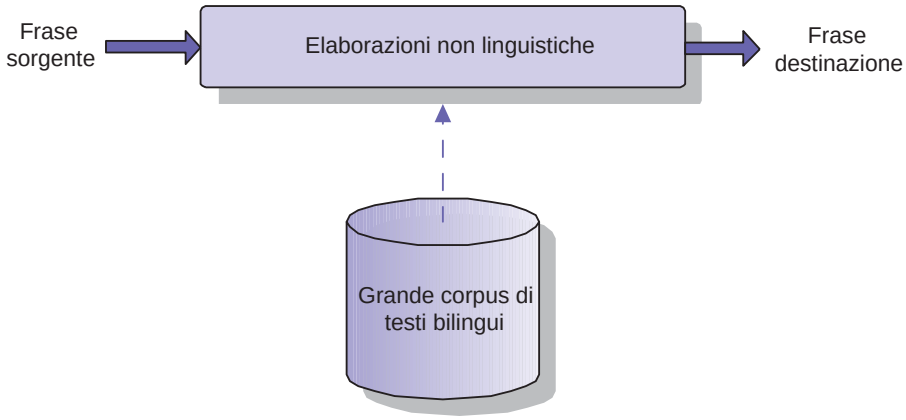


Figura 2.3: Funzionamento dei paradigmi non-linguistic-based

Le principali tipologie di questo nuovo filone sono [10]:

- MT basata su statistica (*Statistical-Based MT - SBMT*)
- MT basata su dialogo (*Dialog-Based MT - DBMT*)
- MT basata su reti neurali (*Neural Network-Based MT - NBMT*)
- MT basata su esempi (*Example-Based MT - EBMT*)

Statistical-Based MT

La produzione di traduzione basata su predizione statistica (SBMT) è fondata su tecniche (derivanti da studi sulla elaborazione del linguaggio parlato) che analizzano statisticamente il corpus parallelo bilingue. In particolare, è utilizzata una variante della regola di Bayes per mostrare che la probabilità che una stringa di parole (T) sia la traduzione di una data stringa di parole (S) è proporzionale al prodotto tra la probabilità che la stringa destinazione sia un'espressione valida

nel linguaggio destinazione e la probabilità che una stringa di parole nel linguaggio sorgente sia la traduzione della stringa di parole nel linguaggio destinazione. Formalmente:

$$P(T|S) \sim P(T) * P(S|T)$$

Se si conoscono le probabilità del membro di destra, si può ottenere la traduzione scegliendo T in modo che la probabilità del membro di sinistra sia massima. Ovviamente, non è possibile conoscere le probabilità di tutte le stringhe nei due linguaggi, pertanto sarà necessario stimarle, definendo modelli probabilistici approssimati costruiti sulle probabilità che possono essere direttamente stimate dai dati esistenti.

Si tratta di un approccio in grado di produrre traduzioni di buona qualità, senza il bisogno di utilizzare grammatiche o lessici. Tuttavia, è necessario che esista un corpus bilingue di notevoli dimensioni, poiché il funzionamento del sistema sarà basato interamente su questo. Inoltre, l'unico modo di migliorare la qualità delle traduzioni è quella di migliorare l'accuratezza dei modelli statistici, che però già per i semplici modelli descritti presentano una quantità incredibile di parametri (nell'ordine dei milioni).

Dialog-Based MT

La traduzione basata su dialogo è un paradigma rivolto a un utente che sia l'autore originale del testo da tradurre. I sistemi basati su questo approccio forniscono un meccanismo per instaurare con questi un dialogo sulla traduzione (un esempio è in figura 2.4), permettendo all'utente stesso di rimuovere le ambiguità del testo di origine e di incorporare dettagli stilistici e culturali, in modo da rendere la traduzione di alta qualità.

```
The word 'pen' means:
        1) a writing pen
        2) a play pen
        3) a pig pen
     NUMBER?>
```

Figura 2.4: Interazione DBMT con l'utente: un esempio

Neural Network-Based MT

Questo è un campo ancora molto sperimentale: sono stati per ora svolti alcuni esperimenti per risolvere mediante la tecnologia delle reti neurali alcuni comuni problemi nell'ambito della Machine Translation, ad esempio l'analisi e la rimozione delle ambiguità lessicali e l'apprendimento di regole grammaticali. Si sono utilizzati però dei vocabolari molto ridotti (poche decine di parole) e una sintassi semplificata, poiché grammatiche e dizionari voluminosi farebbero esplodere la dimensione delle reti neurali e il relativo tempo di apprendimento. Si tratta dunque di un approccio per ora non utilizzabile per costruire un sistema MT realistico.

Example-Based MT

Questo è indubbiamente uno degli ambiti di ricerca che più ha riscosso interesse negli ultimi anni e che ha portato notevoli risultati. Inoltre, è proprio l'approccio EBMT ad avere un ruolo centrale nel progetto presentato in questo lavoro, pertanto ad esso sono dedicate le sezioni seguenti di questo capitolo. In particolare, si cercherà di spiegarne le basi e di mostrarne l'innovatività e le differenze che presenta rispetto ai normali sistemi di traduzione automatica, quelli cioè basati sugli altri paradigmi visti. Si tratta di concetti semplici ma fondamentali per comprendere i veri vantaggi di questo approccio.

2.3 Sistemi EBMT

I sistemi EBMT [10] emulano la traduzione umana, riconoscendo la similarità di una nuova frase nel linguaggio sorgente (da tradurre) con una precedentemente tradotta, ed utilizzando quest'ultima per realizzare una "traduzione per analogia".

L'idea di base dell'EBMT fa riferimento a un database di traduzioni passate, in cui compaiono in parallelo le frasi da tradurre e quelle già tradotte: si tratta della cosiddetta *Translation Memory*. L'accuratezza e la qualità della traduzione dipendono fortemente dalla dimensione e dalla copertura della Translation Memory, ma non è necessario che questa assuma le dimensioni dei database richiesti, ad esempio, dai sistemi di traduzione statistica, poiché non è necessario coprire l'intero vocabolario.

I normali sistemi di traduzione automatica producono le traduzioni applicando conoscenza linguistica; essi analizzano linguisticamente il testo nel linguaggio sorgente, associano le strutture linguistiche alle loro controparti nel testo destinazione, e costruiscono una traduzione da zero. Questi sistemi hanno due principali svantaggi: sono dipendenti dalla lingua poiché impiegano avanzate tecniche linguistiche che si possono applicare esclusivamente a una lingua specifica. Pertanto sono limitati ai linguaggi che possono gestire. Inoltre, poiché un programma per computer non potrà mai riuscire a capire realmente il linguaggio e il contesto, i sistemi automatici producono traduzioni sterili e spesso inopportune. La maggior parte di questi sistemi hanno dizionari e frasi che possono essere personalizzati, ma difficilmente raggiungono la qualità di una traduzione umana professionale.

I sistemi di tipo *Translation Memory (TM)*, invece, memorizzano le traduzioni passate e le propongono quando, traducendo nuovo materiale, incontrano frasi identiche o simili. I sistemi TM "imparano" e migliorano con l'utilizzo: più vengono utilizzati più diventano accurati e utili.

I sistemi TM, insomma, non traducono da soli. Quando il sistema TM è in procinto di tradurre una nuova frase, svolge una ricerca nella Translation Memory e presenta all'utente le traduzioni di frasi identiche o simili che il traduttore ha fatto in passato. Nonostante in molti casi non proponga di suo una traduzione, lo aiuta ad assicurare la consistenza dello stile e della terminologia permettendogli di consultare con facilità le traduzioni passate. In altre parole, i sistemi basati su Translation Memory non traducono ma forniscono consistenza, permettendo al traduttore un lavoro più rapido e di maggiore qualità.

2.3.1 Schema di funzionamento

Lo schema di funzionamento tipico di un sistema EBMT è illustrato in figura 2.5.

La Translation Memory

Al centro di tutto c'è la *Translation Memory* che, come si è visto, contiene il patrimonio di traduzioni che il traduttore professionista ha realizzato in passato; in realtà, per un funzionamento ottimale di questi sistemi, non è necessario inserire in questo DB *tutti* questi documenti. È importante, piuttosto, inserirne

un certo numero che siano il più possibile di argomento attinente ai lavori che si vorranno tradurre in modo assistito. Ad esempio, se il traduttore volesse tradurre un particolare manuale tecnico, sarebbe opportuno che inserisse nella TM, se disponibili, le traduzioni delle passate edizioni dello stesso manuale. In questo modo, infatti, il sistema fornirebbe, dopo aver svolto una ricerca nella TM per frasi identiche o simili, *suggerimenti* più numerosi e mirati, garantendo tra l'altro la consistenza dello stile.

Il meccanismo di *ricerca* delle frasi nella Translation Memory varia da sistema a sistema, così come il modo (in genere molto limitativo) di intendere la "somiglianza" tra esse: gli approcci più utilizzati a questo riguardo verranno illustrati nelle seguenti sezioni di questo capitolo, in cui si descriveranno alcuni dei più importanti sistemi EBMT sviluppati in ambito di ricerca (sezione 2.4.1) e in ambito commerciale (sezione 2.4.3).

Avvenuta la ricerca, il traduttore è in grado di svolgere il proprio lavoro sul documento, utilizzando i preziosi suggerimenti fornitigli dal sistema. In questo modo viene prodotto il documento tradotto e il ciclo si chiude: esso potrà a sua volta essere inserito nella Translation Memory.

Si rende così evidente ancora una volta uno dei grandi pregi dei sistemi EBMT: man mano che passa il tempo ed il sistema viene utilizzato, l'utente può contare su un patrimonio di frasi tradotte sempre più ricco e completo.

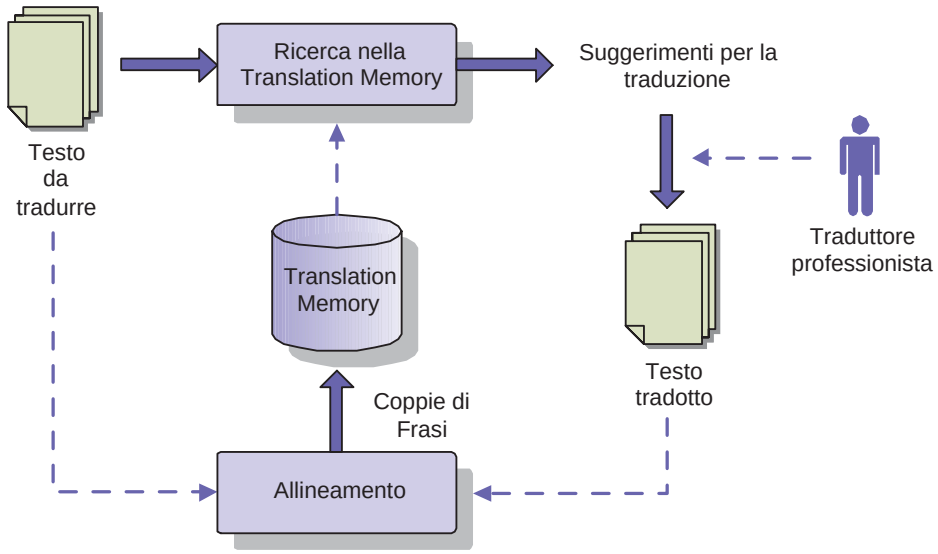


Figura 2.5: Funzionamento di un sistema EBMT

L'allineamento

Rimane da considerare un ultimo aspetto del procedimento, ultimo ma non certo meno importante: l'*allineamento*. Per poter introdurre i documenti nella Translation Memory e perché questa possa essere di una qualche utilità, è necessario che le frasi della lingua sorgente (da tradurre) siano messe in relazione alle corrispondenti frasi tradotte della lingua destinazione. Per poterlo fare è necessario attuare un processo di allineamento che, dati i due documenti nelle due lingue, fornisce in uscita le coppie di frasi da registrare nella TM.

L'allineamento è fondamentalmente una funzione di corrispondenza 1:1 in *spazi bitestuali*. Un *bitesto* (*bitext)* comprende due versioni di un testo, è cioé un testo in due differenti lingue creato dai traduttori ogni volta che traducono un testo. Ogni bitesto definisce uno *spazio bitestuale* di forma rettangolare, come illustrato in figura 2.6.

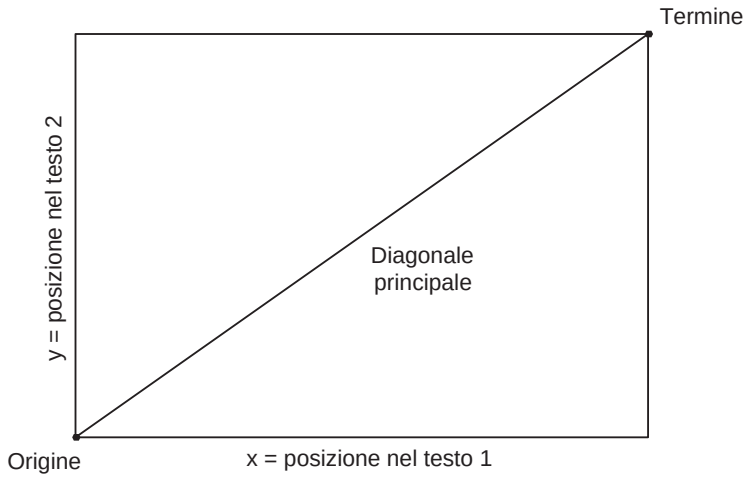


Figura 2.6: Allineamento: uno spazio bitestuale

La larghezza e l'altezza del rettangolo sono le lunghezze delle due componenti di testo. L'angolo del rettangolo in basso a sinistra è l'*origine* dello spazio bitestuale e rappresenta l'inizio dei due testi. L'angolo in alto a destra è il *termine* e ne rappresenta la fine. La linea tra l'origine e il termine è la *diagonale principale*.

Oltre all'origine e al termine, ogni spazio bitestuale contiene un certo numero di *punti di corrispondenza effettivi* (*True Points of Correspondence o TPC*).

Ad esempio se un elemento alla posizione p sull'asse x ed uno alla posizione q sull'asse y sono la traduzione l'uno dell'altro, allora le coordinate (p, q) nello spazio bitestuale rappresentano un TPC. Possono essere definiti TPC anche in corrispondenza di delimitazioni del testo come frasi, paragrafi, capitoli.

L'insieme completo dei punti TPC è l'allineamento vero e proprio: gli algoritmi per ricercare gli allineamenti, come è ovvio, variano anch'essi profondamente da sistema a sistema: nella sezione successiva verranno mostrati gli approcci più diffusi.

2.4 EBMT: Stato dell'arte

In questa sezione verrà analizzato lo stato dell'arte della ricerca scientifica nel campo della traduzione assistita, nonché i più importanti software disponibili in ambito commerciale.

I sistemi di traduzione automatica (HAMT) non sono nella maggior parte dei casi sufficientemente pratici e non possono essere applicati se non in ambiti molto ristretti; altrettanto, sistemi completamente automatici (FAMT) che producano autonomamente traduzioni di alta qualità rimangono a tutt'oggi un sogno. Pertanto, in questa analisi ci si concentrerà sulla Traduzione Umana Assistita dalla Macchina (MAHT). Come si è già ricordato, d'altronde, è proprio questo il campo che ha raccolto i maggiori sforzi dei ricercatori e che ha prodotto i risultati più interessanti.

In particolare ci si concentrerà proprio sui prodotti e sui progetti di Example-based Machine Translation. Dopo aver analizzato i vari approcci, si cercherà anche di evidenziarne le limitazioni e le carenze comuni; esse hanno fornito un notevole spunto per gettare le basi di un progetto che fosse innovativo nel suo campo e che si spingesse oltre la maggior parte di questi limiti: il progetto EXTRA, della cui descrizione ci si occuperà nella seconda parte di questo documento.

2.4.1 La Ricerca scientifica

La somiglianza tra frasi

Il primo a suggerire tecniche di Example-Based Machine Translation è Nagao [26], nel 1984. Questi getta le basi di quella che chiama "Traduzione per analogia": una frase viene semplicemente riconosciuta come simile se compare identica

nei due testi a meno di una (pur simile) parola *di contenuto*, cioè contenente una precisa semantica. La vicinanza nella somiglianza sarebbe in questo caso determinata dalla distanza di significato tra le due parole di contenuto, misurata da metriche basate su un'ontologia. La traduzione risultante non è altro che la sostituzione della diversa parola nella frase più somigliante trovata. Un approccio indubbiamente interessante, ma che come si vede è piuttosto limitato nella ricerca di somiglianza e si appoggia a sofisticate informazioni che non sempre sono disponibili, a meno di non costringere l'utente a una lunga elaborazione dei propri testi.

Altri ricercatori hanno in seguito approfondito l'aspetto della corrispondenza tra le frasi: buoni risultati sono stati ottenuti nell'ambito della ricerca di corrispondenza delle frasi nel loro intero (ad esempio da Foruse e Hida [11]), anche se la maggior parte degli usi dell'EBMT è ristretta a particolari sottoproblemi. Per citare alcuni importanti lavori, Sumita [41] si è occupato della somiglianza tra parole funzionali, Sato [37] di quella tra frasi nominali, sempre Sumita del problema delle frasi preposizionali e dei termini tecnici.

In effetti, su questo particolare argomento non è il caso di soffermarsi ulteriormente: lo stato dell'arte della ricerca scientifica sulla somiglianza tra frasi, alla base dei paradigmi EBMT, non è molto lontano dal corrispondente ambito in Information Retrieval, descritto nel Capitolo 1. Come già sottolineato in quell'occasione, non esiste fino ad ora nessun sistema che proponga un metodo che sia applicabile in generale, non solo a ristretti sottoproblemi necessitanti dati molto dettagliati e spesso non disponibili, e che fornisca suggerimenti mirati, "intelligenti" e non banali (frasi trovate identiche a meno di insignificanti differenze).

L'allineamento

È di un certo interesse soffermarsi sui vari approcci proposti per permettere un allineamento quasi automatico delle frasi da inserire in una Translation Memory, operazione che se fatta manualmente richiederebbe un tempo e una fatica intollerabili per l'utente.

Vi sono molte metodologie per allineare due testi corrispondenti, diverse non solo per il metodo di ricerca delle corrispondenze ma anche per la differente "risoluzione" ottenuta con l'allineamento stesso (vedi figura 2.7). Gale e Church [12] propongono ad esempio un interessante sistema puramente matematico, ba-

sato su distribuzioni di probabilità e sulla lunghezza delle frasi, per allineare le varie frasi all'interno dei paragrafi corrispondenti. Si tratta di un metodo veloce ed efficace nella maggior parte delle situazioni, che però non si occupa dell'allineamento delle entità più piccole di una frase, come ad esempio le parole o i caratteri.

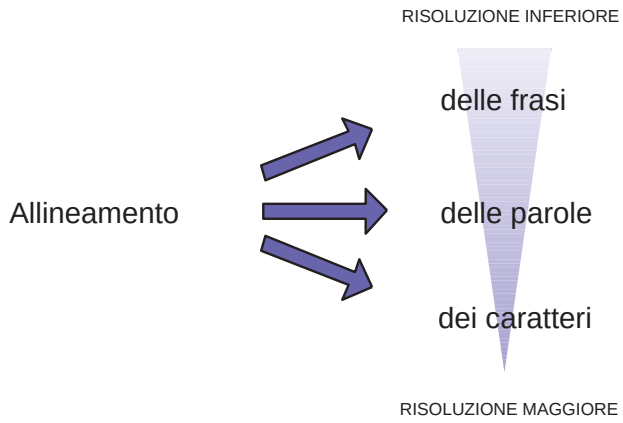


Figura 2.7: Allineamento: le tipologie proposte in ricerca

Sempre Church [6] propone un altro sistema per realizzare un allineamento dei testi nel loro insieme questa volta però a livello dei singoli caratteri (problema studiato anche da Melamed [23]): questo era pensato per risolvere i problemi legati al riconoscimento dei testi derivanti da scansioni (OCR), nei quali sarebbe complesso ricostruire i precisi confini di frasi e paragrafi. Il programma si basa su algoritmi dinamici che allineano gradualmente le varie porzioni dei testi: inizialmente viene prodotto un allineamento sommario su tutto il documento, poi uno sempre più preciso su finestre di testo via via più ridotte. Ciò che più è interessante di questo approccio è la ricerca delle parole *affini (cognates)*: parole che nelle due lingue rimangono uguali o comunque molto simili sintatticamente e che dunque saranno molto probabilmente in corrispondenza. In particolare Church considera affini le parole che contengono 4 caratteri iniziali uguali.

L'allineamento a livello delle parole è proposto da molti altri ricercatori, che nei loro lavori hanno soprattutto tentato di trovare metodi migliori per la ricerca delle parole affini. Simard, Foster e Isabelle [38] propongono di allineare parole corte trovate per intero o lunghe e con almeno 4 caratteri iniziali in comune; alcuni altri ricercatori [17] suggeriscono invece di considerare affini parole con

in comune 4 caratteri consecutivi ma non necessariamente in posizione iniziale. In generale i risultati ottenuti sono piuttosto incoraggianti. Alcuni si spingono ancora più in là, come Tiedemann [44] o Kondrak [18], che utilizza informazioni fonetiche e semantiche per ottenere allineamenti più accurati: ovviamente i tempi di calcolo e la quantità di informazioni necessarie aumentano esponenzialmente.

Come conclusione di questa panoramica, è importante ricordare che l'allineamento è non solo indispensabile per un corretto funzionamento di un sistema EBMT, ma è anche il punto di partenza di elaborazioni successive di notevole interesse. Un esempio di tutto questo sono i lavori di Brew [4] e di Brown [5], che illustrano l'estrazione automatica, a partire dalle coppie di frasi allineate, di un dizionario bilingue. Ovviamente sono necessari notevoli volumi di frasi, ma si tratta senza dubbio di applicazioni di notevole utilità e fascino.

2.4.2 Tipologie di Sistemi commerciali

A livello commerciale, si distinguono tre tipi fondamentali di sistemi per la traduzione assistita MAHT, a seconda del tipo di utente cui si rivolgono [7].

Ambienti software per traduttori professionisti che lavorano in gruppo

Questo tipo di prodotto è rivolto a traduttori competenti che lavorano in gruppo e sono connessi mediante una rete locale. La workstation di ciascun traduttore offre strumenti per:

- accedere a una *Terminologia bilingue*;
- accedere a una *Translation Memory*;
- inviare parti del testo a un server per la Machine Translation.

Spesso questi strumenti integrano un elaboratore di testi e associano scorciatoie da tastiera ai termini e alle frasi trovate nella Terminologia e nella Translation Memory. Una importante decisione di progetto riguarda se offrire uno specifico word processor o se invece appoggiarsi a uno di quelli sviluppati da terze parti. Normalmente, il server supporta strumenti per:

- gestire il *Database Lessicale Multilingue Condiviso* (MLDB);
- gestire un database di terminologia multilingue (MTDB);

- gestire una Translation Memory condivisa (in questo frangente, sono cruciali un accesso concorrente e delle rigide procedure di validazione);
- gestire i compiti di traduzione.

Ambienti software per traduttori professionisti indipendenti

Questi ambienti sono in genere meno potenti, molto più economici e accessibili da tutti gli elaboratori di testi e di documenti commerciali. Per quanto riguarda i dizionari, non esiste più un MLDB centrale da gestire, ma è comunque molto importante per i traduttori poter facilmente creare, accedere, modificare, esportare e importare la terminologia e le frasi dai database.

Strumenti per traduttori occasionali

Un traduttore occasionale potrebbe essere competente in entrambe le lingue ma anche solo nella lingua sorgente, quella cioè dei testi da tradurre. Di fatto, esistono degli strumenti per aiutare persone con una scarsa conoscenza della lingua di destinazione a produrre traduzioni parametrizzabili.

Questi strumenti offrono funzionalità diverse da quelle dei professionisti:

- la Translation Memory è molto ridotta, se non addirittura assente;
- i dizionari devono contenere molti termini generali e spesso esistono tre livelli di dizionari: termini personali, terminologia e vocabolario generale;
- ci sono aiuti per la lingua di destinazione: thesaurus, coniugatori, controllori di stile, ecc.

2.4.3 I Sistemi commerciali più diffusi

I sistemi in commercio, appartenenti alle tre categorie viste, sono molto numerosi. Esempi di prodotti per traduttori professionisti sono *Trados*, *Atril Déjà Vu*, *Cypresoft Trans Suite*, *IBM Translation Manager*, *SITE-Eurolang Eurolang Optimizer*; importanti prodotti per traduttori occasionali sono invece *Ambassador* di Language Engineering Comp. o *Power Translator* di Globalink.

Tuttavia, come ci si sarà resi conto dalla precedente descrizione, la categoria di strumenti per traduttori occasionali è solo ai margini dell'argomento centrale di questa panoramica, la traduzione EBMT per esempi. Le categorie che più si

rivelano interessanti da analizzare, ai fini di evidenziarne le funzionalità offerte, sono pertanto quelle dei software per traduttori professionisti. I software più diffusi e di maggiore rilevanza in questo settore sono tre, e verranno brevemente presentati nei paragrafi seguenti.

Trados Team Edition

Trados 5 Team Edition [45] unisce in un ambiente integrato una notevole serie di funzionalità, quali:

- gestione di Translation Memory;
- gestione di database di terminologia;
- allineamento di frasi;
- gestione e modifica dei documenti tradotti o da tradurre;
- sofisticate funzionalità di project management.

Il programma centrale di questo ambiente è il cosiddetto *Translator Workbench* (vedi figura 2.8), in cui il traduttore può effettuare la traduzione di un documento con l'ausilio dei suggerimenti forniti dal motore di ricerca. La ricerca nei database è strutturata su vari livelli:

- Livello 1: Ricerca di frasi intere. La translation memory viene scandita alla ricerca di frasi esattamente identiche (*exact match*) o quasi (con un piccolo numero di parole diverse, *fuzzy match*).
- Livello 2: Ricerca di una parte. Selezionando la parte interessata della frase da tradurre, viene ricercato un elemento della Translation Memory che sia identico a questa.
- Livello 3: Ricerca di terminologia. Il database di terminologia viene esaminato per trovare delle ulteriori corrispondenze esatte.

Occorre inoltre sottolineare come il modulo di allineamento sia uno dei più potenti sul mercato: esso fornisce una interfaccia per l'allineamento manuale ma utilizza anche una serie di algoritmi proprietari per eseguire un allineamento automatico delle frasi di alta qualità.

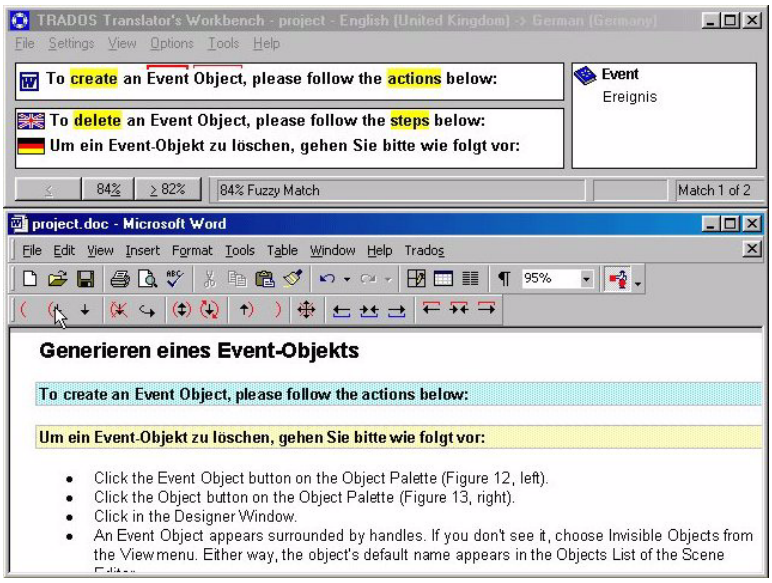


Figura 2.8: Sistemi commerciali: Trados Translator Workbench

Atril Déjà Vu

Déjà Vu [9] è un prodotto integrato che utilizza tecniche di EBMT e Translation Memory. È anch'esso in grado di cercare e fornire suggerimenti su frasi trovate in modo esatto *(exact match)* o approssimato *(fuzzy match)*; utilizza inoltre una serie di database aggiuntivi che vengono chiamati *memory database, terminology database e project lexicon* per migliorare la qualità delle frasi suggerite.

Déjà Vu ha anche molte caratteristiche aggiuntive: è un ambiente integrato per la traduzione (in figura 2.9 l'interfaccia utilizzata), completo di word processor proprietario, ricerca automatica delle frasi presenti nella translation memory, possibilità di presentare in vario modo i documenti da tradurre, visualizzando ad esempio le frasi ordinate alfabeticamente o nel loro contesto, possibilità di lavorare con diversi traduttori su un unico progetto multilingua, supporto ad una moltitudine di formati di file ed altro ancora.

La ricerca di frasi simili viene svolta per passi successivi, con queste modalità:

1. *Exact Scan*: viene scandita la Translation Memory alla ricerca di frasi identiche

2. *Fuzzy Scan*: vengono ricercate frasi *quasi* uguali a quelle da tradurre (identiche ma con un alcune differenze in parole non fondamentali come sigle, nomi propri, ecc.)

3. *Assemble*: se la ricerche precedenti non sono sufficienti, è possibile attivare questa nuova modalità che cerca di tradurre una frase unendo varie sottoparti trovate in modo *esatto ed integrale* nella TM.

Infine, per quanto riguarda l'allineamento, è fornito un apposito programma aggiuntivo che, pur non offrendo un procedimento automatico, dispone di un'interfaccia utente grafica per permettere al traduttore di svolgere questo delicato lavoro.

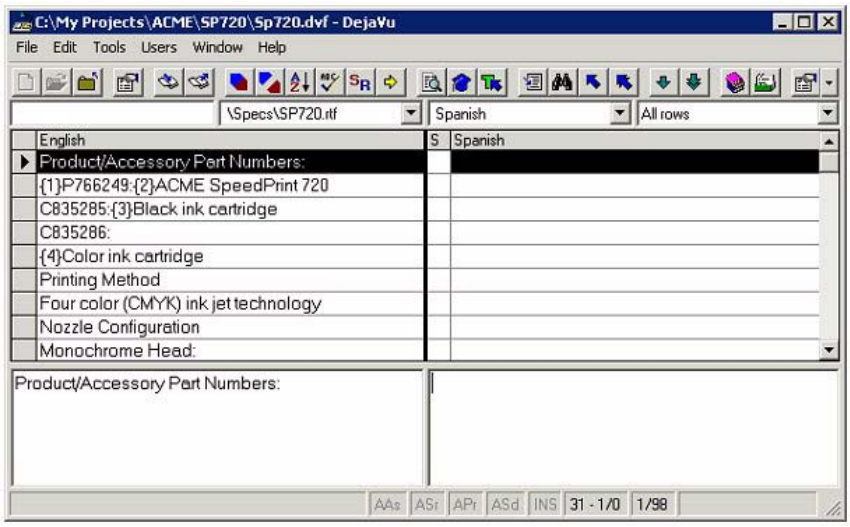


Figura 2.9: Sistemi commerciali: Atril Déjà Vu

Cypresoft Trans Suite 2000

Si tratta di una suite di programmi creata da traduttori per i traduttori [46]. I prodotti inseriti in questa suite sono:

- *Editor*, il programma cuore del sistema (figura 2.10), che unisce in un'unica interfaccia la modifica delle frasi sorgente e destinazione, dei dizionari e la visualizzazione dei suggerimenti di traduzione.
- *Align*, il programma di allineamento delle frasi.
- Utility per la gestione dei dizionari: *Dictionary Setup Utility (DSU), Master Dictionary Editor e Translation Management System (TSMS)*, quest'ultimo in grado di estrarre informazioni statistiche dai dizionari e dalla Translation Memory.

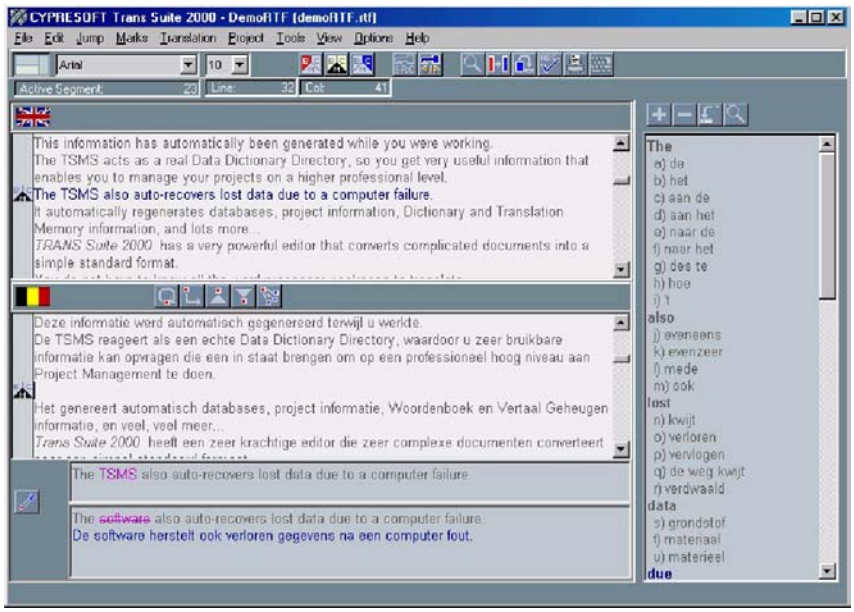


Figura 2.10: Sistemi commerciali: Cypresoft Trans Suite 2000 Editor

Le caratteristiche del motore di ricerca delle frasi sono piuttosto tradizionali:

- Ricerca di frasi intere esatte (*Exact match*)
- Ricerca di frasi intere simili, che utilizza il cosiddetto *Cypresoft Fuzzy Logic Generation 2 engine*: come detto per Dèjà Vu, vengono proposte frasi con alcune differenze in parole non fondamentali come date, orari, sigle, nomi propri, ecc. Inoltre, queste differenze vengono evidenziate graficamente con colori particolari.

L'allineatore è anche in questo caso praticamente manuale: le frasi dei due documenti vengono spezzate e il traduttore deve indicare i giusti collegamenti tra frasi sorgente e destinazione mediante una semplice interfaccia grafica (vedi figura 2.11).

2.4.4 I limiti attuali

I vari ambienti analizzati sono abbastanza analoghi come caratteristiche e, purtroppo, anche come limiti. Le limitazioni riscontrate possono essere riassunte nei seguenti punti fondamentali:

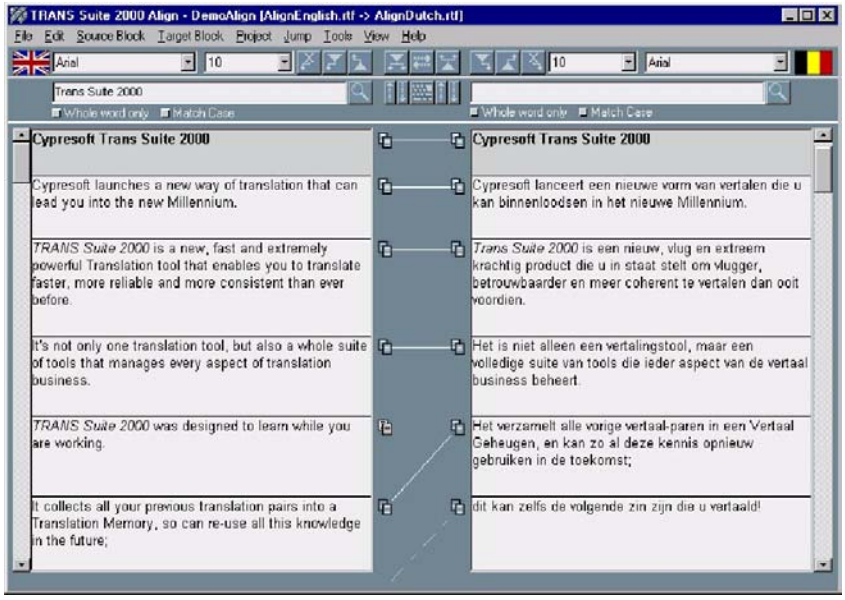


Figura 2.11: Sistemi commerciali: Cypresoft Trans Suite 2000 Align

- *Fuzzy match spesso inefficaci.* La ricerca di frasi simili non è basata su algoritmi adeguatamente potenti e flessibili (problema riscontrato anche in [21]), ma si riduce, come già evidenziato, alla ricerca di frasi essenzialmente identiche, sia nella struttura sia nel significato (vedi figura 2.12). È evidente che quando è possibile trovare questi suggerimenti, sicuramente le nuove traduzioni vengono notevolmente velocizzate; più realisticamente, però, capita che nella Translation Memory non si trovino frasi *così* simili, ed è allora che i limiti di queste modalità di ricerca diventano evidenti.

Figura 2.12: Fuzzy match: un esempio

- *Ricerca di sottoparti fortemente incompleta.* Quei prodotti che offrono algoritmi di ricerca non solo di frasi intere ma anche di sottoparti, lo fanno in un modo lacunoso ed insoddisfacente. Non solo ci si restringe in questo caso ai soli match esatti, ma si ricercano soltanto quelle sottoparti che sono

state inserite *come tali* nella TM: se una frase inserita *per intero* è simile solo in una sua sottoparte, quest'ultima non viene recuperata e proposta. Un limite molto grave, che costringe il traduttore ad inserire nella TM non solo la coppia di frasi tradotta ma anche, individualmente, le varie sottoparti che riterrà poter tornare utili.

- *Netta separazione tra Terminologia e Translation Memory.* Come diretta conseguenza dell'inabilità nel ricercare autonomamente le sottoparti delle frasi nella TM, il traduttore deve trattare separatamente anche l'immisione della terminologia specifica in dizionari aggiuntivi, nonostante queste siano informazioni già presenti nelle frasi della TM.

- *Allineamenti non automatici.* La procedura di allineamento delle frasi è quasi sempre di tipo manuale, costringendo il traduttore a lunghe sessioni di lavoro sui testi paralleli per evidenziare le frasi, metterle in corrispondenza, ecc. Inoltre, nessun tipo di allineamento è previsto all'*interno* delle frasi, ad esempio tra le varie parole.

Come già ricordato, partendo proprio dalla comprensione di questi notevoli limiti sono state gettate le basi del progetto EXTRA, innovativo nelle concezioni e pensato per superare la maggior parte di essi. Nella parte seguente di questo documento ci si occuperà della sua descrizione, dall'approccio adottato alla progettazione, dalle caratteristiche fondamentali ai risultati ottenuti.

Parte II

Il progetto EXTRA

Capitolo 3

Ricerca di similarità tra frasi

In questo capitolo ci si concentrerà sull'approccio seguito per realizzare quella che è la funzione alla base di tutto il progetto: la ricerca di similarità tra frasi. Essa merita una descrizione dettagliata e verrà analizzata preliminarmente anche perché, come si è visto dalla prima parte, riguarda un argomento di ricerca di per sé di notevole interesse in svariati ambiti.

Nel capitolo seguente, si descriverà invece l'ambiente EXTRA nel suo insieme: si analizzeranno le ulteriori funzionalità di quest'ultimo, collocando quanto precedentemente descritto nel contesto globale e mostrando quelle caratteristiche aggiuntive che lo rendono un sistema completo di Example-Based Machine Translation.

3.1 Le finalità

Progettando gli algoritmi di ricerca di similarità si è cercato di creare dei procedimenti che soddisfacessero i seguenti requisiti:

- *Rigorosità*: pur riguardando un campo che ha molti elementi soggettivi, basarsi su una solida struttura teorica, che fornisca un metodo chiaro ed univoco per quantificare la somiglianza tra frasi.
- *Praticità*: non utilizzare informazioni che vanno oltre le frasi stesse, spesso non disponibili o non pratiche da ottenere (come strutture che ne descrivono la semantica).
- *Flessibilità*: adattarsi alle diverse tipologie di utenti e/o testi.

- *Efficacia*: produrre dei risultati effettivamente utili e di qualità.

- *Efficienza*: essere ottimizzate anche come tempi di esecuzione, indispensabile per poter gestire notevoli volumi di dati.

Per soddisfare tutti questi punti, si è studiato e implementato un procedimento piuttosto innovativo, basato nelle sue linee di fondo sul concetto di *Edit Distance* e su un'analisi della frase prettamente sintattica. Il procedimento di ricerca vero e proprio è suddiviso in più fasi, ognuna implementata mediante query ad un DBMS. Per migliorare l'efficienza, si utilizzano inoltre particolari strutture dati (le tabelle dei *q-grammi*) ed una serie di *filtri* per ottenere ottime prestazioni in ciascuna di esse.

A seconda delle esigenze dell'utente, è possibile ricercare intere frasi simili ma anche sottoparti di queste, permettendo in questo modo un maggiore sfruttamento del potenziale insito nella Translation Memory: spesso due frasi sono diverse nel loro insieme, ma presentano magari una sottoparte del tutto analoga come struttura e, pertanto, utile. Come si vedrà, è anche possibile agire sui principali parametri in modo, ad esempio, da escludere frasi che siano *diverse* da quella cercata oltre una certa soglia, da aumentare l'accuratezza o piuttosto la velocità della ricerca, e così via.

3.2 I passi fondamentali del procedimento

Come si ricorderà dal Capitolo 2 e in particolare dalla sezione 2.3, la ricerca di similarità tra frasi è al centro del funzionamento di un sistema EBMT: dato un documento di testo da tradurre, questa operazione consente di accedere alla base di dati di frasi tradotte (la Translation Memory), identificare le frasi più *simili* a quelle contenute nel documento e proporle come suggerimento per la futura traduzione (figura 3.1).

In questo capitolo, ci disinteresseremo di tutte le operazioni, pur indispensabili, relative alla creazione e alla gestione della Translation Memory, come l'identificazione e la corretta separazione delle frasi, l'allineamento tra le frasi nelle lingue sorgente e destinazione, e così via: come ricordato esse verranno approfondite nel capitolo seguente.

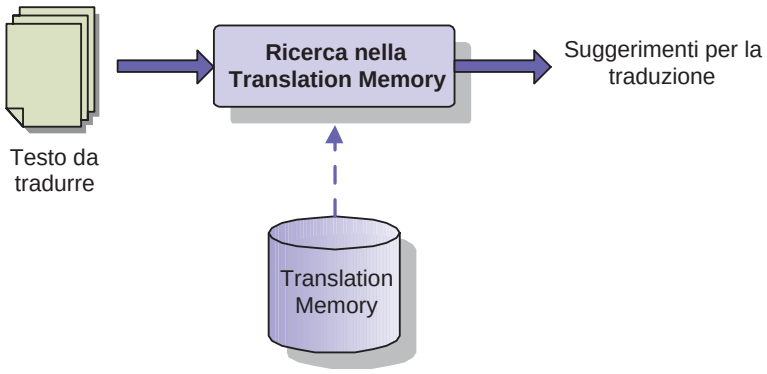


Figura 3.1: La ricerca di similarità

Ciò che invece occorre qui dettagliare è il modo in cui viene affrontata la ricerca delle frasi simili.

Immaginando di disporre già di una Translation Memory opportunamente organizzata (con frasi già estratte, allineate ed accoppiate nelle due lingue), nonché delle frasi da tradurre già adeguatamente separate, il processo prevederà i seguenti passi (vedi figura 3.2):

1. preparazione delle frasi da tradurre
2. ricerca di somiglianza vera e propria

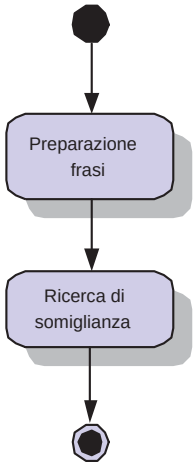


Figura 3.2: Activity diagram della ricerca di similarità

Aggiungendo dettagli a questo schema di fondo (figura 3.3):

- la prima fase comprende in realtà l'elaborazione delle frasi da tradurre e il loro inserimento in apposite tabelle della base di dati;
- la ricerca di somiglianza può essere suddivisa in ricerca delle frasi intere simili e ricerca delle sottoparti simili (quest'ultima opzionale).

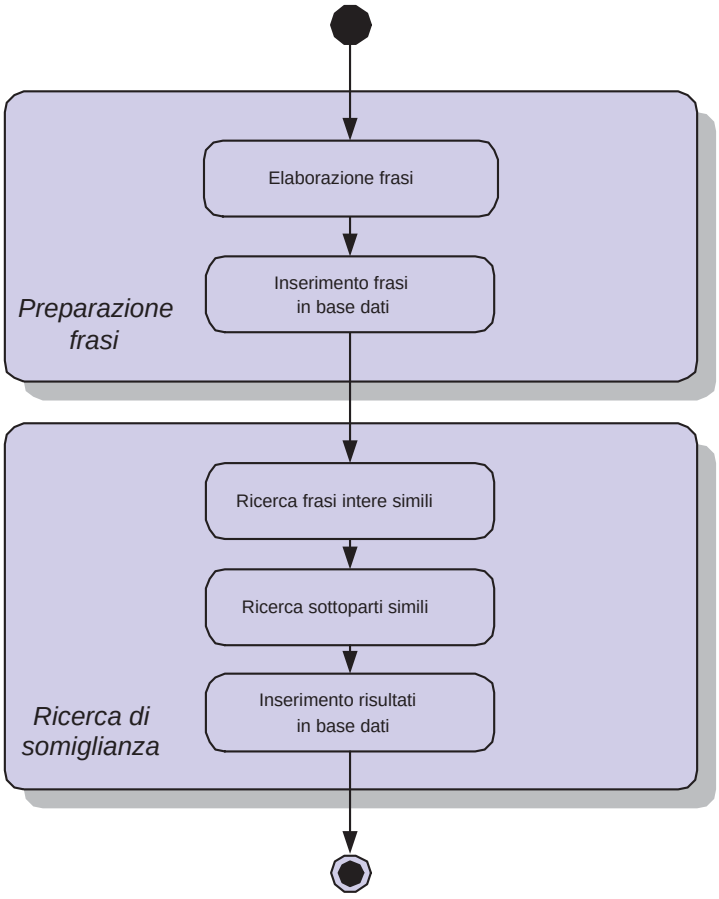


Figura 3.3: Activity diagram dettagliato

Nelle sezioni seguenti verranno spiegati in dettaglio tutti questi punti.

3.3 Le tabelle utilizzate

Vengono qui presentate le tabelle utilizzate nel procedimento di ricerca di similarità, poiché si tratta di informazioni indispensabili alla comprensione del

funzionamento delle query in esso utilizzate; nello schema si evidenziano in particolare i nomi delle tabelle, gli attributi, le *primary key* e le *foreign key* (le frecce rappresentano i collegamenti tra le tabelle referenziate), nonché gli *indici* utilizzati. Un avvertimento: nonostante sia possibile comprendere sin d'ora il significato e l'utilizzo della maggior parte degli attributi, per alcuni (ad esempio quelli contenenti dati di allineamento) questo sarà del tutto chiaro solo dopo aver letto le relative sezioni del presente capitolo e del successivo.

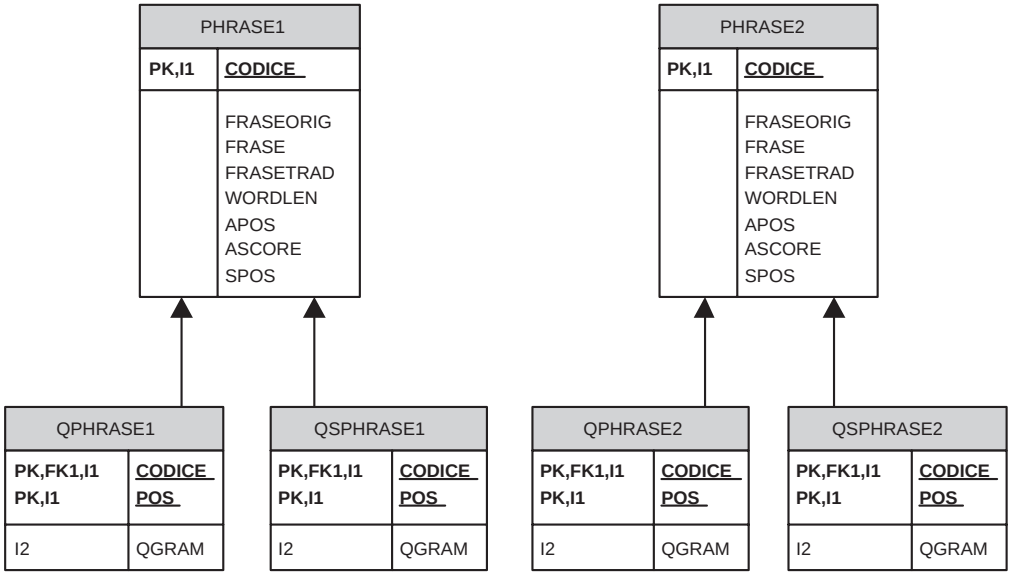


Figura 3.4: Schema delle tabelle: tabelle delle frasi

In figura 3.4 vengono mostrate le tabelle utilizzate per la memorizzazione delle frasi (i nomi sono quelli predefiniti ma sono configurabili dall'utente):

- *Phrase1* contiene le frasi della translation memory, cioè quelle dei documenti di riferimento, da cui verranno estratte le frasi simili da suggerire (tabella permanente);
- *Phrase2* contiene le frasi del documento da pretradurre (tabella temporanea).

Le due tabelle hanno la medesima struttura di fondo; in particolare per ciascuna frase vengono memorizzate le seguenti informazioni:

- *Codice*: un codice univoco, che costituisce la chiave primaria delle tabelle

- *FraseOrig*: la frase vera e propria nella sua forma originaria (prima cioè dell'elaborazione)
- *Frase*: la frase elaborata (tipicamente, come si vedrà, l'elaborazione consiste nello stemming)
- *WordLen*: la lunghezza (in parole) della frase elaborata
- *SPos*: le coordinate di allineamento dell'elaborazione (stemming)

e, solo per la Translation Memory:

- *FraseTrad*: la traduzione della frase
- *APos*: le coordinate di allineamento delle parole
- *AScore*: i punteggi di allineamento delle parole.

Inoltre, nelle tabelle *QPhrase1* e *QSPhrase1*, *QPhrase2* e *QSPhrase2*, vengono memorizzati i q-grammi (sezione 3.4.2) delle suddette frasi, utilizzati per accelerare e filtrare la ricerca delle frasi simili. I campi utilizzati sono i seguenti:

- *Codice*: il codice della frase cui il q-gramma si riferisce
- *Pos*: la posizione del q-gramma nella frase
- *Q-gram*: la stringa vera e propria del q-gramma, su cui è previsto anche un indice per migliorare le prestazioni di ricerca.

In figura 3.5 vengono presentate anche le tabelle progettate per contenere i risultati delle ricerche di somiglianza; *FullMatch* contiene le coppie di frasi simili trovate (e relativa distanza), mentre *MatchPos* e *SubMatch* contengono, rispettivamente, i risultati intermedi e finali della ricerca delle sottoparti di frase simili (oltre ai codici delle frasi interessate esse contengono anche le posizioni di inizio e fine della sottoparte). Per una descrizione dettagliata dell'utilizzo di queste tabelle si rimanda alle relative sezioni: query di ricerca delle frasi intere (3.6) e query di ricerca delle sottoparti (3.9).

Nella sezione B.1, in appendice, sono riportati gli script completi utilizzati nella creazione di queste tabelle.

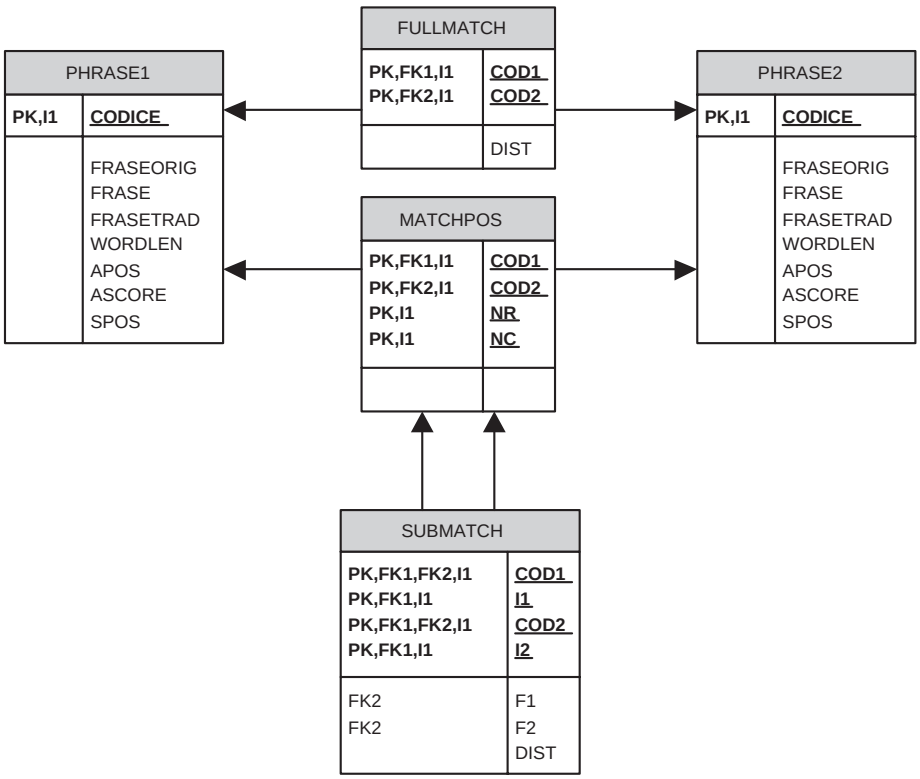


Figura 3.5: Schema delle tabelle: tabelle dei risultati

3.4 Preparazione delle frasi

La ricerca delle frasi simili ha al suo cuore, per lo meno nella fase iniziale di ricerca di frasi intere, una rielaborazione della tecnica di *approximate string join* descritta in [14] e basata sul concetto di q-gramma [42, 43, 48]; per poter applicare questa tecnica efficientemente è necessario innanzitutto elaborare le frasi da ricercare, quindi generare i *q-grammi* associati, inserire le frasi e i q-grammi nelle opportune tabelle, quindi utilizzare queste ultime per la ricerca.

3.4.1 L'elaborazione

Al fine di rendere la successiva operazione di ricerca più efficiente ed efficace, è necessario elaborare preliminarmente le frasi. In particolare, a seconda delle preferenze dell'utente, sono possibili due differenti processi:

- semplice rimozione della punteggiatura
- *stemming* (o *normalizzazione*)

Rimuovendo la punteggiatura è possibile lasciare la frase inalterata ma prepararla ad una corretta individuazione delle parti simili.

Lo *stemming* è invece un processo più complesso, che si occupa, come visto nella sezione 1.6.1, di eliminare parole molto comuni e poco significative (le cosiddette *stopwords*, ad esempio gli articoli e le preposizioni) e di portare le altre in una forma standard (i nomi vengono ricondotti al singolare, i verbi all'infinito, e così via). Ad esempio, la frase:

"This means you can collect a library of images"

diviene

"mean can collect library image".

In questo modo la ricerca di frasi simili prescinde dalla particolare forma delle parole e si può concentrare solo su quelle che veramente danno significato alla frase. Per effettuare lo stemming, si fa uso di un package svilippato da F. Gavioli per la lingua inglese [13], che è stato ottimizzato e lievemente modificato per meglio integrarsi in questo progetto. Poiché non rientrante direttamente nell'argomento affrontato in questo capitolo, la struttura di questo package, nonché le modifiche apportate, verranno descritte nei capitoli successivi.

3.4.2 La suddivisione in q-grammi

Il concetto di *q-gramma* per una stringa di testo (come descritto in [14]) è quello di breve sottostringa di lunghezza q caratteri consecutivi; i vari q-grammi associati a una stringa σ si ottengono allora facendo scorrere una finestra ampia q caratteri dall'inizio alla fine della stringa. Poiché i q-grammi all'inizio e alla fine della stringa avrebbero meno di q caratteri di σ, vengono introdotti nuovi caratteri "#" e "$" e si estende concettualmente la stringa σ aggiungendo un prefisso di *q-1* caratteri "#" e un suffisso di *q-1* caratteri "$".

Esprimendo formalmente queste nozioni:

Definizione 3.1 ***Q-gramma posizionale.*** *Sia Σ una alfabeto finito di dimensione $|\Sigma|$, sia σ una stringa di lunghezza n, $\sigma[i...j], 1 \leq i \leq j \leq n$ una sottostringa di σ di lunghezza j-i+1 che comincia alla posizione i. Si definisce q-gramma posizionale di una stringa σ la coppia $(i, \sigma[i...i+q-1])$, dove*

$\sigma[i...i+q-1]$ è il q-gramma di σ che inizia alla posizione i, contando sulla stringa estesa. L'insieme G_σ di tutti i q-grammi posizionali di una stringa σ è l'insieme di tutte le $|\sigma|+q-1$ coppie ottenute da tutti i q-grammi di σ.

L'intuizione che sta dietro all'uso dei q-grammi per la ricerca di somiglianza tra le frasi è che due frasi simili avranno un gran numero di q-grammi in comune. Ai nostri fini, questi concetti sono stati applicati, ma con una modifica fondamentale: l'unità di misura non è più il singolo *carattere* di una stringa, ma la *parola* di una frase. In questo modo un q-gramma non sarà più una sequenza di q caratteri, ma di q parole.

Ad esempio, data la frase "you can collect a library of images", l'insieme corrispondente di q-grammi posizionali (con q=3) è il seguente:

{(1,"# # you"), (2,"# you can"), (3,"you can collect"),
(4,"can collect a"), (5,"collect a library"), (6,"a library of"),
(7,"library of images"), (8,"of images $"), (9,"images $ $") }.

In particolare, il procedimento di ricerca di similarità adottato prevede l'estrazione di due set di q-grammi, caratterizzati da diversi valori di q:

- *q* standard, utilizzato nella ricerca delle frasi intere
- *qSub*, un q inferiore (tipicamente unitario) più adatto invece alla ricerca delle sottoparti

3.4.3 L'inserimento

Le frasi (nella doppia versione originale ed elaborata, ad esempio con stemming) e i q-grammi, dopo essere stati elaborati ed estratti nel modo visto, vengono registrati nelle tabelle già presentate nella sezione 3.3.

3.5 La metrica di similarità: Edit distance

Per poter spiegare il metodo seguito nella ricerca, è necessario innanzitutto chiarire il concetto di *edit distance*. Perché i suggerimenti di traduzione proposti dal programma possano essere di una qualche utilità, è necessario poter ricercare

non solo frasi da proporre *esattamente* uguali a quelle da tradurre, ma anche solo *approssimativamente* uguali. È dunque necessario ricercare i *match* tra frasi in modo approssimato: per poterlo fare è necessario definire una *metrica* di approssimazione. Esistono molte proposte teoriche per esprimere l'idea di "uguaglianza approssimata"; tra queste la nozione di *edit distance* tra due frasi è quella che ci è sembrata più adatta ai nostri fini, nonché quella che poteva meglio essere inserita nel contesto di un DBMS relazionale (come mostrato anche in [14]).

3.5.1 Definizione classica

Secondo la nozione classica, riguardante le stringhe di caratteri, l'*edit distance* tra due stringhe è definita come la sequenza di operazioni che, con il minor costo, trasforma una stringa nell'altra.

Formalmente: si indichino stringhe arbitrarie con $\sigma_1, \sigma_2, ...$, lettere dell'alfabeto con *a,b,c...*, la lunghezza della stringa σ con $|\sigma|$, l'i-esimo carattere di σ con $\sigma(i)$ ($i \in \{1..|\sigma|\}$) e la stringa vuota con ε.

Definizione 3.2 ***Edit distance.*** *L'edit distance ed(σ_1, σ_2) tra due stringhe σ_1 e σ_2 è il minimo costo della sequenza di operazioni che trasforma σ_1 in σ_2. Il costo della sequenza di operazioni è la somma dei costi delle operazioni individuali. Le operazioni sono un insieme finito di regole nella forma $\delta(\sigma_3, \sigma_4) = c$, dove σ_3 e σ_4 sono due stringhe diverse e c è un numero reale non negativo. Una volta che l'operazione ha convertito una sottostringa σ_3 in σ_4, nessuna ulteriore operazione può essere svolta su σ_4.*

Nel nostro caso si considera il seguente insieme di operazioni possibili, ciascuna di costo unitario:

- *Inserimento*: $\delta(\varepsilon, a)$, cioè inserire la lettera a
- *Cancellazione*: $\delta(a, \varepsilon)$, cioè cancellare la lettera a
- *Sostituzione*: $\delta(a, b)$, cioè sostituire a con b

Alla luce di queste definizioni, allora, l'edit distance tra le due stringhe rappresentanti, ad esempio, le parole "surgery" e "survey" sarà di 2 unità: una sostituzione (da "g" a "v") e una cancellazione ("r").

3.5.2 L'algoritmo per il calcolo

Il più semplice algoritmo per risolvere il problema del calcolo dell'edit distance è quello che è stato riscoperto più volte nel passato, per l'applicazione alle più diverse aree [36, 49, 19]. Si fa qui riferimento alla versione presentata in [27].

L'algoritmo è basato sulla programmazione dinamica. Immaginando di dover calcolare $ed(\sigma_1, \sigma_2)$, si riempie una matrice $C_{0..|\sigma_1|,0..|\sigma_2|}$, dove $C_{i,j}$ rappresenta il minimo numero di operazioni necessarie per trasformare $\sigma_1(1..i)$ in $\sigma_2(1..j)$. Il calcolo procede come segue:

$$\begin{array}{lcll} C_{i,0} & = & i & \\ C_{0,j} & = & j & \\ C_{i,j} & = & C_{i-1,j-1} & se(\sigma_1(i) = \sigma_2(j)) \\ & & 1 + min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) & altrimenti \end{array}$$

Il risultato finale si legge nella cella in basso a destra: $C_{|\sigma_1|,|\sigma_2|} = ed(\sigma_1, \sigma_2)$. L'algoritmo deve riempire la matrice in modo che le celle in alto, a sinistra e in alto a sinistra rispetto alla cella corrente siano calcolate prima di quella cella; è possibile perciò utilizzare sia un riempimento per righe da sinistra verso destra, sia per colonne dall'altro verso il basso. La complessità di questo algoritmo sarà $O(|\sigma_1||\sigma_2|)$, sia per il caso medio sia per il caso peggiore.

Per vederne l'applicazione all'esempio visto precedentemente, in figura 3.6 viene mostrato il calcolo di *ed("surgery", "survey")*.

		S	U	R	G	E	R	Y
	0	1	2	3	4	5	6	7
S	1	**0**	1	2	3	4	5	6
U	2	1	**0**	1	2	3	4	5
R	3	2	1	**0**	1	2	3	4
V	4	3	2	1	**1**	2	3	4
E	5	4	3	2	2	**1**	**2**	3
Y	6	5	4	3	3	2	2	**2**

Figura 3.6: Edit distance: esempio di calcolo

In neretto compare il cammino verso il risultato finale: spostamenti in basso, a destra e in basso a destra corrispondono rispettivamente a inserimenti, cancellazioni, e uguaglianze (o sostituzioni). Il risultato è 2 (cella in basso a destra).

3.5.3 Applicazione alle frasi

I concetti visti sono indubbiamente interessanti, ma per essere applicabili al nostro contesto è necessario utilizzare un cambio di punto di vista: le operazioni di edit non riguarderanno tanto i *caratteri*, ma le *parole* di una frase.

Così, la edit distance tra le due frasi "The tools disk contains some disk utilities" e "The tools disk includes some utilities" (mostrate nell'esempio di figura 3.7) è uguale a 2: in questo caso 2 *parole* di distanza.

		The	*tools*	*disk*	*contains*	*some*	*disk*	*utilities*
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
The	*1*	**0**	1	2	3	4	5	6
tools	*2*	1	**0**	1	2	3	4	5
disk	*3*	2	1	**0**	1	2	3	4
includes	*4*	3	2	1	**1**	2	3	4
some	*5*	4	3	2	2	**1**	**2**	3
utilities	*6*	5	4	3	3	2	2	**2**

Figura 3.7: Edit distance: applicazione alle frasi

Come si vede, l'algoritmo di calcolo in sé è immutato e vengono mantenute tutte quelle caratteristiche e proprietà che anche in questo ambito si rivelano di notevole interesse.

In particolare, ai nostri fini è fondamentale la possibilità di quantificare in modo preciso e univoco la "somiglianza" tra due frasi.

Non solo: senza nessuna operazione aggiuntiva è possibile stabilire la frase che *più* è simile a una data. Quella con il punteggio di distanza minore sarà insomma la miglior candidata, quella che potrà comparire per prima nella lista dei suggerimenti: come si vede l'applicazione dell'edit distance fornisce di suo

un ottimo metodo per stabilire il *ranking*, che come si è visto nel Capitolo 1 è indispensabile in ambito di Information Retrieval.

Inoltre, altro punto fondamentale, questo punteggio di somiglianza è anche sensibile all'*ordine* delle parole, non solo alla loro presenza: è evidente che le frasi "The wolf eats a sheep" non è per niente equivalente a "A sheep eats the wolf", pur contenendo le stesse identiche parole: il significato, anzi, è agli antipodi. Mentre i tradizionali modelli di IR in questo caso reputerebbero identiche le due frasi, l'approccio con edit distance restituisce una notevole distanza, facendo intuire la scarsa somiglianza tra i due concetti.

3.5.4 L'algoritmo ottimizzato per diagonali

Come mostrato da E. Ukkonen [47], è possibile migliorare l'algoritmo precedentemente descritto per quanto riguarda il caso peggiore. È possibile sfruttare la proprietà che le diagonali della matrice di programmazione dinamica sono monotonamente crescenti. In particolare, $C_{i+1,j+1} \in \{C_{i,j}, C_{i,j} + 1\}$.

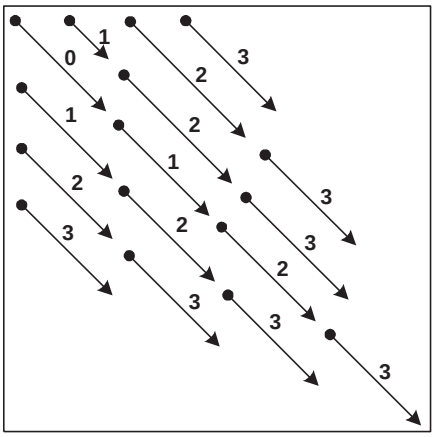


Figura 3.8: Edit distance: l'algoritmo ottimizzato per diagonali

Come mostrato in figura 3.8, l'algoritmo procede calcolando tratti (*stroke*) diagonali; ogni tratto rappresenta un numero di errori ed è una sequenza in cui entrambe le stringhe (frasi) corrispondono.

Quando inizia un tratto di e errori, esso continua finché continuano gli adiacenti tratti *e-1 oppure* finché continua la corrispondenza delle stringhe (è sufficiente *una* delle condizioni enunciate) (figura 3.9).

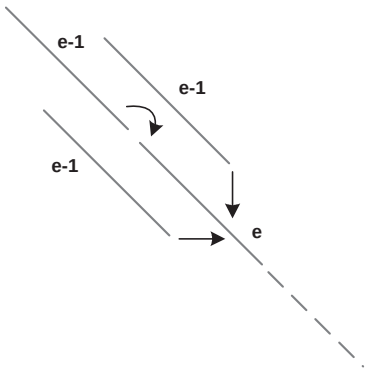


Figura 3.9: Edit distance: il calcolo dei tratti diagonali

In questo modo è possibile arrivare alla cella finale (il risultato) calcolando un numero notevolmente inferiore di celle e velocizzando dunque i calcoli: in figura 3.10 viene mostrato l'esempio completo del calcolo dell'edit distance con il metodo ottimizzato, applicato alle frasi già utilizzate negli esempi della sezione precedente.

		The	*tools*	*disk*	*contains*	*some*	*disk*	*utilities*
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
The	*1*	**0**	1					
tools	*2*	1	**0**	1				
disk	*3*		1	**0**	1			
includes	*4*			1	**1**	2		
some	*5*				2	**1**	**2**	
utilities	*6*					2	2	**2**

Figura 3.10: Edit distance: il calcolo ottimizzato

Nell'implementazione, sono stati inoltre apportati i seguenti accorgimenti:

- raggiunta la massima distanza consentita (k), l'algoritmo si interrompe comunicando esito negativo;
- durante il calcolo dei tratti diagonali, in caso di raggiungimento degli estre-

mi della matrice il calcolo procede solo più per i tratti utili (quelli cioè che porteranno verso la cella finale) (vedi figura 3.11).

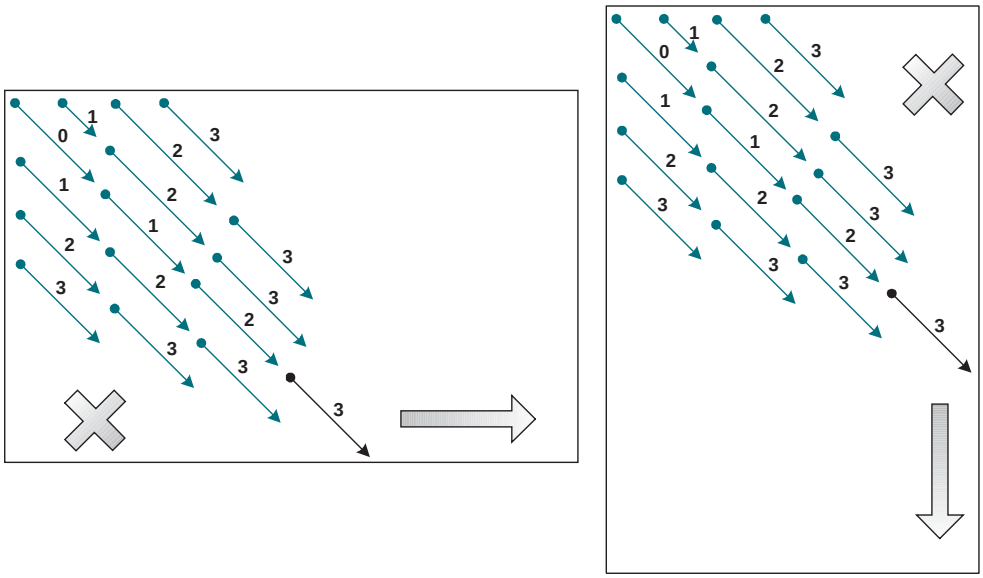


Figura 3.11: Edit distance: accorgimenti utilizzati nell'algoritmo

3.5.5 La scelta degli algoritmi

Gli algoritmi enunciati per il calcolo dell'edit distance vengono realizzati con opportune *stored procedure* che vengono richiamate direttamente dal DBMS nelle query di ricerca. In particolare, per i nostri fini, si è deciso di utilizzare entrambi gli algoritmi mostrati:

- quando non è richiesta dall'utente la ricerca delle sottoparti simili, viene utilizzato l'algoritmo per diagonali;
- in caso contrario, si vedrà che tutti i dati della matrice sono necessari e perciò in questo caso l'algoritmo utilizzato è quello standard.

3.6 La query di ricerca delle frasi intere simili

La query per la ricerca è una rielaborazione di quanto presentato in [14] e presenta la seguente struttura:

```
INSERT INTO FULLMATCH
SELECT r2.codice AS cod2, r1.codice AS cod1,
        wordEditDistanceDiag (r1.frase, r2. frase, <k>)
FROM <tab1> r1, <qtab1> r1q, <tab2> r2, <qtab2> r2q
WHERE r1.codice= r1q.codice
AND r2.codice = r2q.codice
AND r1q.qgram = r2q.qgram
 ... AND <<filtri>> ...
AND wordEditDistanceDiag (r1.frase, r2.frase, <k>) >= 0
```

Si tratta di una query costruita dinamicamente, in cui i nomi delle tabelle sono variabili e in cui figurano parametri liberamente modificabili (come k). Come si può vedere, essa agisce sulle frasi individuando le coppie simili ed inserendone i dati direttamente nella tabella *FullMatch* (codici identificativi della coppia di frasi e relativa distanza).

WordEditDistanceDiag() è la *stored procedure* che implementa l'algoritmo per il calcolo dell'edit distance ottimizzato per diagonali descritto nella sezione 3.5.4; in particolare verifica se le due frasi passate come parametro sono sufficientemente "simili", cioè con una edit distance inferiore a k. In caso affermativo, restituisce un valore non negativo (la distanza calcolata, che verrà registrata in tabella), altrimenti restituisce *-1* e provoca l'esclusione della coppia di frasi candidate.

La sezione identificata dall'etichetta *<<filtri>>* contiene particolari condizioni che riescono a filtrare preventivamente le possibili coppie di frasi, velocizzando così l'esecuzione della query. Essi si basano sull'utilizzo dei q-grammi precedentemente estratti e memorizzati e verranno ora analizzati in dettaglio.

3.6.1 I filtri

Un punto di forza dell'approccio presentato è la possibilità di inserire nella query una serie di *filtri* che riducono al minimo indispensabile il richiamo della *stored procedure* per il calcolo dell'edit distance; essi, sfruttando i q-grammi memorizzati nelle tabelle ausiliarie, riescono a garantire ottime prestazioni, grazie al basso numero di *falsi positivi* restituiti, e correttezza, grazie all'assenza di *false esclusioni*. I filtri implementati per questa query sono l'adattamento di quelli descritti in [14]; segue una descrizione del fondamento teorico di ciascuno di

essi, oltre a come vengono implementati nella query precedentemente vista. Si ricorda che ciascuno di essi è liberamente attivabile o disattivabile mediante opportuni parametri, in modo da poterne testare accuratamente e singolarmente il comportamento.

Il filtro di conteggio

L'idea di fondo del *filtro di conteggio* è di sfruttare l'informazione degli insiemi di q-grammi G_{σ_1} e G_{σ_2}, ignorando le informazioni posizionali, per determinare se le frasi σ_1 e σ_2 sono all'interno di un'edit distance k (come sempre, si ricorda che nella nostra applicazione lunghezza di una frase e distanze sono misurate in *parole*).

Enunciato 3.1 *Si considerino le frasi σ_1 e σ_2, di lunghezza $|\sigma_1|$ e $|\sigma_2|$ rispettivamente. Se σ_1 e σ_2 sono entro un'edit distance di k, allora la cardinalità di $G_{\sigma_1} \cap G_{\sigma_2}$, ignorando le informazioni posizionali, deve essere di almeno $max(|\sigma_1|, |\sigma_2|) - 1 - (k-1) * q$, dove q è la dimensione dei q-grammi.*

La query si arricchisce allora del seguente predicato:

GROUP BY r2.codice, r1.codice, r1.frase, r2.frase
HAVING COUNT(∗) >= (r1.wordlen − 1 − (<k> − 1) ∗ <q>)
AND COUNT(∗) >= (r2.wordlen − 1 − (<k> − 1) ∗ <q>),

In esso, così come in quelli degli altri filtri, si può notare l'utilizzo della colonna *wordlen*, contenente la lunghezza in parole della frase; si è preferito registrarla in tabella come dato derivato piuttosto che calcolarla di volta in volta con una funzione esterna poiché in quest'ultimo caso le numerose chiamate avrebbero ridotto notevolmente le prestazioni. $<q>$ rappresenta il numero di parole in un q-gramma (la sua dimensione) ed è un parametro liberamente modificabile.

Il filtro di posizione

Il filtro di posizione complementa il filtro di conteggio tenendo conto dell'informazione posizionale contenuta in un q-gramma. L'enunciato su cui si fonda è il seguente:

Enunciato 3.2 *Se le stringhe σ_1 e σ_2 sono entro una edit distance di k, allora un q-gramma posizionale in una non può corrispondere a un q-gramma posizionale nell'altra che disti da esso più di k posizioni.*

Nella query questo filtro è implementato con un'ulteriore clausola where:

WHERE ...
AND ABS (r1q.pos − r2q.pos) <= <k>,

dove *ABS* è la funzione SQL che restituisce il valore assoluto dell'espressione passata.

Il filtro di lunghezza

Si può considerare infine che osservando la lunghezza delle frasi confrontate è possibile scartare svariate coppie di frasi candidate:

Enunciato 3.3 *Se due frasi σ_1 e σ_2 sono entro una edit distance k, le loro lunghezze non possono differire per più di k.*

Nella query si aggiungerà allora:

WHERE ...
AND ABS (r1.wordlen − r2.wordlen) <= <k>

3.6.2 Edit distance relativa e arrotondamenti

Un'ulteriore aggiunta alla query è legata alla seguente considerazione: è facile rendersi conto della scarsa flessibilità di un parametro k (massima distanza consentita) assoluto, non legato alle frasi cercate. In particolare, ad esempio, questo significa ammettere un uguale numero di errori nel caso in cui la frase cercata sia di 3 o di 20 parole. Questo non è ovviamente desiderabile; per ovviare a questo inconveniente, si è deciso di utilizzare un parametro che definisse non tanto il numero massimo *assoluto* di errori permessi, ma il numero di errori *relativo* alla lunghezza della frase cercata.

Il nuovo k non sarà più un intero, bensì un numero decimale: ad esempio impostando k=0.4, il numero massimo di errori ammessi ricercando le frasi simili

a una di 10 parole sarà 4, per una frase di 20 sarà invece 8, e così via. Nel controllo dell'edit distance e nei relativi filtri, si utilizzerà allora un "*k equivalente*" ottenuto moltiplicando il k così definito per la lunghezza della frase cercata:

*<k> * r2.wordlen*

In questo modo, semplicemente sostituendo al k assoluto il valore di questo prodotto, si mantiene una piena compatibilità con i filtri e con il controllo sull'edit distance, fornendo al tempo stesso risultati più accurati.

Un ulteriore accorgimento riguarda il caso in cui questo prodotto non sia un numero intero: utilizzando direttamente il valore decimale ottenuto, ad esempio verificare se "*<edit distance> ≤ 2.9*", equivale in realtà a verificare "*edit distance ≤ 2*" (intero inferiore), essendo la distanza sempre intera. Si vede che l'errore commesso è in questo caso molto maggiore che non arrotondando il valore: nel caso in esempio, l'arrotondamento sarebbe per eccesso e il controllo avverrebbe su una distanza massima consentita di 3, più vicina al valore richiesto.

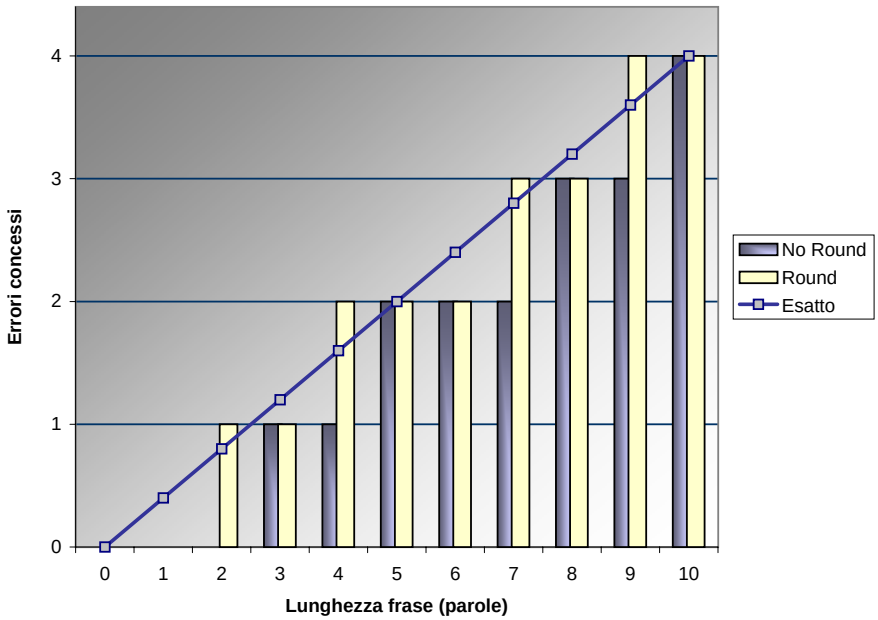


Figura 3.12: Arrotondamento delle edit distance relative

Questo concetto è visualizzato, per k=0.4, nel grafico di figura 3.12: esso mostra gli andamenti del numero di errori concessi al variare della lunghezza

della frase. Come si vede, nel caso di arrotondamento si commette un errore inferiore.

Nella query si utilizza pertanto il seguente "*K equivalente*":

*ROUND(<k> * r2.wordlen),*

calcolato partendo dal parametro decimale k modificabile dall'utente e arrotondato mediante la funzione SQL *ROUND()*.

3.6.3 La query completa

La query completa di tutti questi accorgimenti è perciò la seguente:

```
INSERT INTO FULLMATCH
SELECT r2.codice AS cod2, r1.codice AS cod1,
        wordEditDistanceDiag (r1.frase, r2. frase, <k>)
FROM <tab1> r1, <qtab1> r1q, <tab2> r2, <qtab2> r2q
WHERE r1.codice = r1q.codice
AND r2.codice = r2q.codice
AND r1q.qgram = r2q.qgram
                                        -- filtro di posizione
AND ABS (r1q.pos - r2q.pos) <= ROUND(<k> * r2.wordlen)

                                        -- filtro di lunghezza
AND ABS (r1.wordlen - r2.wordlen)
          <= ROUND(<k> * r2.wordlen)

                                        -- filtro di conteggio
GROUP BY r2.codice, r1.codice, r1.frase, r2.frase
HAVING COUNT(*) >= (r1.wordlen - 1
                    - (ROUND(<k> * r2.wordlen) - 1) * <q>)
AND COUNT(*) >= (r2.wordlen - 1
                    - (ROUND(<k> * r2.wordlen) - 1) * <q>)

AND wordEditDistanceDiag (r1.frase, r2.frase, <k>) >= 0
```

3.7 Oltre le frasi intere

3.7.1 I limiti da superare

La query di ricerca di frasi intere presentata è indubbiamente molto potente. Grazie ai filtri e all'utilizzo dei q-grammi garantisce una notevole efficienza, senza per questo rinunciare all'efficacia: essendo anzi basata sul concetto di edit distance, è in grado di trovare *match* tra frasi di notevole utilità.

Una tecnica perfetta? Non in tutto e per tutto: così com'è stata presentata, ha la limitazione di fondo di ridurre i match alle sole frasi intere. Pertanto, se ci si fermasse a questo stadio, l'algoritmo di ricerca presenterebbe gli stessi limiti degli algoritmi utilizzati nei programmi EBMT commerciali: a meno di non inserire nella translation memory non solo le frasi intere ma anche alcune delle sottoparti ritenute più significative, queste non verrebbero mai suggerite.

È indubbiamente una limitazione notevole, poiché una frase della Translation Memory potrebbe essere utile solo parzialmente: pur essendo diversa nel suo insieme, una sua sottoparte potrebbe essere molto simile alla frase cercata. Spingendosi ancora oltre: una sottoparte di una frase contenuta nella TM potrebbe essere simile a una sottoparte della frase cercata.

Riuscire a sfruttare al massimo il potenziale della Translation Memory, come si vede, non è certo un compito facile, soprattutto se si tiene conto che è indispensabile che gli algoritmi utilizzati nella ricerca mantengano buone prestazioni anche con notevoli volumi di dati. Dopo numerosi affinamenti e ottimizzazioni, sono state progettate per questo lavoro una serie aggiuntive di query in grado di:

- ricercare match di similarità tra *qualunque* sottoparte delle frasi da pretradurre e *qualunque* sottoparte delle frasi della Translation Memory, senza che queste siano state precedentemente estratte;
- ricercare sottoparti non solo in modo esatto ma anche in modo *approssimato*, con la stessa metrica di similarità utilizzata per le frasi intere.

Tutto questo senza *mai* accantonare l'efficienza. L'utilità di tutto questo è evidente: tra i suggerimenti al traduttore compariranno non più solo frasi intere ma anche le frasi contenenti delle sottoparti simili, con le relative sottoparti interessate. Come si vedrà nel prossimo capitolo, queste sottoparti rappresenteranno

dei veri e propri suggerimenti di traduzione poiché insieme ad essi verrà proposta, grazie agli algoritmi di allineamento, la corrispondente parte di traduzione.

3.7.2 Le nuove modalità di ricerca

La ricerca delle frasi intere ha un funzionamento che è schematizzato in figura 3.13: attingendo ai dati della Translation Memory (in questo caso le tabelle delle frasi e dei q-grammi) registra nella tabella dei risultati *FullMatch* i dati dei match relativi alle frasi intere simili trovate (codice della frase da pre-tradurre cercata, codice della frase della TM simile ad essa e distanza).

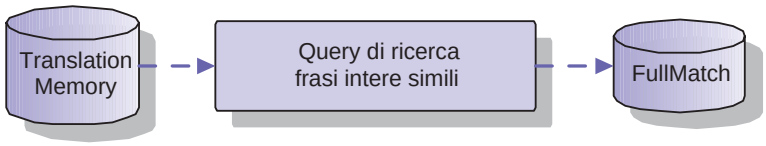


Figura 3.13: Funzionamento della ricerca (solo frasi intere)

Questo meccanismo viene ora esteso alla ricerca delle sottoparti simili: nella figura 3.14 è mostrato il nuovo schema di funzionamento.

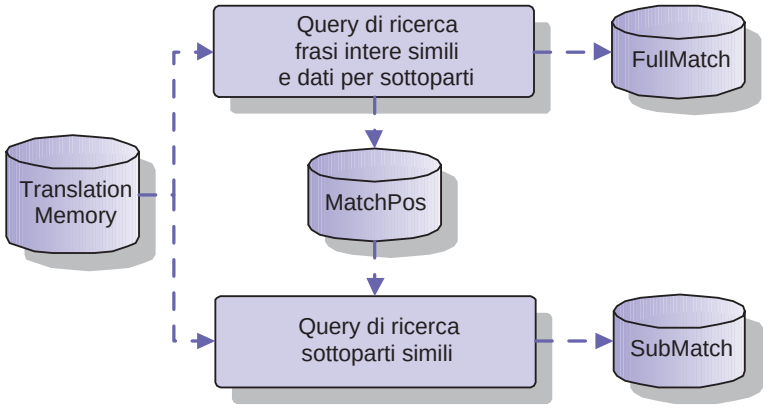


Figura 3.14: Funzionamento della ricerca (esteso alle sottoparti)

Come si vede, il primo passo di ricerca non si limita più a ricercare i match tra le frasi complete, ma registra anche delle informazioni aggiuntive in una ulteriore tabella dei risultati: *MatchPos*. Nella sezione seguente si spiegherà in dettaglio

quali sono queste informazioni e i diversi modi che sono stati messi a disposizione per ricavarle.

Registrati questi risultati intermedi nel DB, la preparazione alla seconda parte della ricerca è conclusa ed è quindi possibile eseguire la query di ricerca delle sottoparti vera e propria: attingendo alla TM e ai dati intermedi, questa registrerà in un'ulteriore tabella dei risultati (*SubMatch*) i match relativi alle migliori sottoparti simili trovate.

3.8 Preparazione alla ricerca delle sottoparti

3.8.1 Le informazioni aggiuntive da estrarre

Nella esecuzione della query di ricerca per frasi intere (sezione 3.6) si è visto che, per tutte le coppie di frasi che soddisfano le condizioni dei vari filtri, viene calcolata la matrice di distanza per poter stabilire l'edit distance tra le due frasi in questione. Con alcune modifiche, e senza rilevanti cali di prestazioni, è possibile estrarre da questi calcoli delle informazioni che si riveleranno preziose per la successiva ricerca di sottoparti: le posizioni nelle due frasi di eventuali parole uguali (mostrate con E in figura 3.15).

		The	*tools*	*disk*	*contains*	*some*	*disk*	*utilities*
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
The	*1*	**0 E**	1	2	3	4	5	6
tools	*2*	1	**0 E**	1	2	3	4	5
disk	*3*	2	1	**0 E**	1	2	**3 E**	4
includes	*4*	3	2	1	1	2	3	4
some	*5*	4	3	2	2	**1 E**	2	3
utilities	*6*	5	4	3	3	2	2	**2 E**

Figura 3.15: I dati utili per la successiva ricerca di sottoparti simili

L'algoritmo per il calcolo dell'edit distance prevede già la costruzione della tabella e il confronto tra tutte le coppie di parole (sezione 3.5.2): l'unica diffe-

renza in questo caso è che non potremo utilizzare l'ottimizzazione per diagonali e che non saremo più interessati solo alla cella in basso a destra con il risultato, ma anche a quelle caselle in corrispondenza a parole uguali. In particolare, per le coppie di frasi che soddisfano determinate condizioni, nella tabella *MatchPos* verranno registrati:

- i *codici* delle frasi in questione
- le *posizioni* delle parole uguali, che non sono altro che le coordinate della cella interessata

Pertanto, nelle frasi dell'esempio in figura 3.15, verranno registrate informazioni del tipo: "Codice frase1, Codice frase2, 1, 1" (prima parola della prima frase uguale alla prima parola della seconda), "Codice frase1, Codice frase2, 2, 2" (seconda parola della prima frase uguale alla seconda parola della seconda), e così via.

3.8.2 Le tre versioni delle query

Sono state sviluppate tre versioni (figura 3.16) di questa prima parte della ricerca, che in modi diversi raggiungono la stessa finalità: quella di eseguire la ricerca delle frasi intere e di preparare la successiva ricerca delle sottoparti.

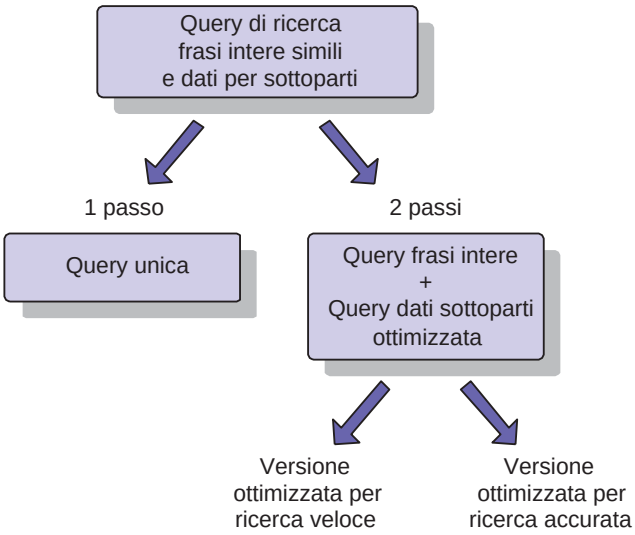


Figura 3.16: Le tre versioni della ricerca

Si tratta di rielaborazioni ed estensioni della query utilizzata per la ricerca delle frasi intere (già presentata nella sezione 3.6), nei casi più complessi costituite da due diversi passi di interrogazione successivi. Sarà l'utente a scegliere, per un dato lavoro di pretraduzione, il metodo più adatto alle particolari esigenze.

Versione a un solo passo

Questo è indubbiamente il metodo concettualmente più semplice: la query presentata per la ricerca delle sole frasi intere viene modificata ed estesa all'estrazione delle informazioni aggiuntive necessarie. La query è sostanzialmente invariata nella struttura vista nella sezione 3.6, con la differenza che il controllo sull'edit distance, invece di essere svolto tramite la stored procedure che esegue il calcolo ottimizzato per diagonali

wordEditDistanceDiag (r1.frase, r2. frase , <k>) >= 0

viene svolto mediante una nuova funzione:

wordEditDistanceSubCheck (r1.frase, r1.codice, r2. frase , r2.codice, <k>) >= 0.

Quest'ultima è perfettamente compatibile con la precedente, permettendo così l'individuazione dei match tra frasi intere, ma rinuncia all'ottimizzazione per diagonali per fornire invece l'individuazione e la registrazione nel DB delle parole uguali trovate, che serviranno per la successiva elaborazione.

In questo modo, con una piccola variante è possibile utilizzare una query unica che in un solo passo ricerca i match interi e prepara alla successiva ricerca di submatch. Tuttavia è un approccio che spesso si rivela poco conveniente: questo perché la query presenta una serie di filtri che sono stati costruiti pensando alle frasi intere. Questo significa che mantenendoli accesi si perderebbero gran parte delle possibili sottoparti simili (una coppia di frasi diverse nel loro insieme ma con una sottoparte simile potrebbe venire scartata dai filtri prima di passare al calcolo dell'edit distance), mentre disattivandoli si rinuncerebbe all'efficienza. Per ovviare a questi inconvenienti sono state pensati i metodi a 2 passi.

Versione a 2 passi, veloce

Le versioni a due passi si basano sulla seguente considerazione: poiché la query delle frasi intere presenta una serie di filtri che la rendono parzialmente incompatibile, a meno di un decadimento nelle prestazioni, con la ricerca delle sottoparti,

è opportuno eseguirla *inalterata* come primo passo. Una sua versione rielaborata verrà eseguita questa volta non in suo luogo ma *successivamente*, con il compito di registrare le informazioni necessarie alla successiva query delle sottoparti. La query modificata presenta in questo caso filtri differenti, adatti in questo caso alle sottoparti:

WHERE r1.codice = r1q.codice
AND r2.codice = r2q.codice
AND r1q.qgram = r2q.qgram
−− nuovo filtro di ridondanza
AND r2.codice **NOT** IN (**SELECT** cod2
FROM FULLMATCH)

−− nuovo filtro di conteggio
GROUP BY r2.codice, r1.codice, r1.frase, r2.frase
HAVING COUNT(∗) >= <minCount>

AND wordEditDistanceSubCheck (r1.frase, r1.codice, r2.frase,
r2.codice, <k>) >= 0

Come sempre, la registrazione delle informazioni aggiuntive è svolta internamente alla stored procedure richiamata in ultimo. Come si vede il nuovo *filtro di ridondanza* evita di analizzare ulteriormente tutte quelle coppie frasi che già nel primo passaggio sono state riconosciute come simili: in questo caso le frasi sono simili nella loro interezza, pertanto ulteriori analisi alle relative sottoparti sarebbero ridondanti.

Inoltre, la nuova versione del filtro di conteggio risulta compatibile con la ricerca di sottoparti. Il minimo conteggio necessario *minCount* è cosi calcolato:

$$minCount = lMinSub - 1 - (Round(kSub * lMinSub) - 1) * q - 2 * (q - 1)$$

In pratica si tratta della formula del filtro di conteggio standard (sezione 3.6.1) calcolata però, invece che di volta in volta sulla lunghezza delle frasi, una volta per tutte per il caso peggiore: quello della lunghezza minima *lMinSub*. Questo è infatti il parametro che regola la lunghezza minima delle sottoparti che verranno successivamente ricercate. Inoltre, si utilizza $kSub$ (massima distanza

relativa consentita per le sottoparti) in luogo di k, e compare il fattore correttivo $-2 * (q - 1)$.

Nel suo insieme, questa formula calcola il nuovo numero minimo di q-grammi uguali che le due frasi candidate devono presentare per poter avere una sottoparte simile di lunghezza almeno $lMinSub$. Per farlo tiene conto e concilia diversi aspetti: la dimensione dei q-grammi (q) e quella minima della sottoparte ($lMinSub$) sono variabili e indipendenti, inoltre non si può più tener conto dei q-grammi estesi (in tutto $2 * (q - 1)$), che sono disponibili solo per le frasi intere.

Giocando su questi parametri, la versione qui presentata viene denominata *veloce* poiché utilizza nella query il set di q-grammi di dimensione maggiore (normalmente avente $q = 3$), quello utilizzato anche nella query del primo passaggio: in questo modo il DBMS, lavorando sulle stesse tabelle, fornisce una notevole ottimizzazione di prestazioni. Inoltre, poiché tipicamente la dimensione dei q-grammi maggiori è paragonabile alla lunghezza minima delle sottoparti (normalmente anche $lMinSub = 3$), il valore minCount risulta piuttosto basso, cioè uguale a uno o inferiore: normalmente si considerano pertanto le frasi che abbiano almeno 1 q-gramma, cioè q parole consecutive, in comune.

Nonostante si possa vedere che anche in questo caso non tutte le possibili sottoparti vengono individuate (per grandi valori di q e bassi valori di $lMinSub$ non è detto che ci siano q-grammi comuni), tuttavia si vede che con le impostazioni standard e con tipiche Translation Memory, già è possibile individuare buona parte (80% circa) dei suggerimenti possibili, in un tempo molto ridotto.

Versione a 2 passi, accurata

Questa versione è sostanzialmente analoga a quella a due passi vista precedentemente, tuttavia essa si prefigge lo scopo di trovare *tutti* i possibili risultati intermedi per la successiva ricerca delle sottoparti.

La query sostanzialmente non varia: quello che varia è il set di q-grammi utilizzato (e il relativo q): in questo caso vengono utilizzati dei q-grammi di dimensione inferiore ($qSub$), per permettere una ricerca più "fine". Mentre con il filtro di conteggio della versione precedente si richiedeva un piccolo numero di grandi q-grammi in comune, ora si richiede un numero maggiore di q-grammi più piccoli. In particolare, utilizzando i valori di default ($qSub = 1$ e $lMinSub = 3$), tipicamente si ricercano frasi con almeno due parole (q-grammi di dimensio-

ne unitaria) in comune, anche non consecutive, non scartando in questo modo nessuna coppia di frasi potenzialmente utile.

Grazie alla piena flessibilità dei parametri, non si è però costretti a mantenere $qSub = 1$: per valori maggiori di $lMinSub$ sarà possibile impostarlo a valori adeguatamente superiori senza rinunciare all'accuratezza. Per la precisione, si può dimostrare che la piena correttezza dei filtri basati sui q-grammi di dimensione $qSub$ è assicurata se tale valore $qSub$ appartiene al seguente intervallo:

$$qSub \in \left[1, \left\lfloor \frac{lMinSub}{ROUND(kSub * lMinSub) + 1} \right\rfloor\right]$$

Per quanto riguarda la velocità di esecuzione: sebbene le prestazioni di questa versione non possano evidentemente competere con la precedente, il modo in cui è strutturata consente comunque una notevole efficienza.

Si ricorda che in appendice B.3 sono riportate le query complete.

3.9 La query di ricerca delle sottoparti simili

3.9.1 Il funzionamento

Dopo aver registrato nella tabella *MatchPos* le posizioni delle parole uguali e i codici delle relative frasi, è possibile eseguire la query che effettua la vera e propria ricerca delle sottoparti.

Il compito di questa query è piuttosto impegnativo: cercare i "segmenti" di frase aventi inizio e fine corrispondenti alle parole uguali estratte dalla tabella *MatchPos* e che rispondano alle seguenti caratteristiche:

- abbiano una lunghezza minima ($lMinSub$);
- siano effettivamente simili, cioè presentino una distanza relativa massima $kSub$;
- non siano contenuti in altri segmenti più ampi e che soddisfino a loro volta la condizione precedente.

Un compito veramente arduo, anche perché le entry nella tabella *MatchPos* da cui costruire tutte le possibili combinazioni di segmenti sono tipicamente diverse decine di migliaia. Per consentire buone prestazioni pur non rinunciando

a nessuna di queste caratteristiche, anche in questa query sono stati utilizzati una serie di filtri studiati ad hoc.

Per capire il funzionamento della query, occorre tenere presente che:

- Come per la query delle frasi intere, le tabelle $< tab1 >$ ($r1$) e $< tab2 >$ ($r2$) rappresentano, rispettivamente, la tabella delle frasi della Translation Memory e del documento da pretradurre.
- Dalla tabella *MatchPos* vengono estratte le "coordinate" dei punti corrispondenti a parole uguali (come visto nella sezione 3.8.1): in particolare vengono estratte due posizioni, *m1* e *m2*, rispettivamente punto di inizio e di fine del segmento simile da esaminare.
- Immaginando, come spiegato nella sezione 3.8.1, la matrice delle frasi con ciascuna colonna corrispondente alle parole della frase cercata (*r2*) e ciascuna riga a quelle della frase della Translation Memory (*r1*), ognuno di questi punti è individuato da due valori: *nc* e *nr*, rispettivamente numero di colonna e di riga.

3.9.2 La query

La struttura della query è la seguente:

```
INSERT INTO SUBMATCH
  SELECT m1.cod2, m1.nc, m2.nc, m1.cod1, m1.nr, m2.nr,
      wordEditDistanceDiag (wordSubString(r1.frase, m1.nr, m2.nr),
      wordSubString(r2.frase, m1.nc, m2.nc), <kSub>)
  FROM MATCHPOS m1, MATCHPOS m2, <tab1> r1, <tab2> r2
  WHERE m1.cod1 = m2.cod1
  AND m1.cod2 = m2.cod2
  AND m1.cod1 = r1.codice
  AND m1.cod2 = r2.codice
                              -- controllo estremi
  AND m1.nr < m2.nr
  AND m1.nc < m2.nc
                              -- controllo lunghezza minima
  AND (m2.nc - m1.nc + 1) >= <lMinSub>
  AND (m2.nr - m1.nr + 1) >= <lMinSub>
```

AND <<filtri di lunghezza, conteggio e posizione>>

AND wordEditDistanceDiag (wordSubString(r1.frase, m1.nr, m2.nr),
wordSubString(r2.frase, m1.nc, m2.nc), <kSub>) >= 0

AND << filtro di inclusione >>

La query lavora in modo simile a quella delle frasi intere: mentre questa registrava nella tabella *FullMatch* i codici delle frasi simili e la relativa distanza (minore di k), questa registra nella tabella *SubMatch* i codici delle frasi contenenti i migliori segmenti simili, le loro coordinate (posizione delle parole) e la relativa distanza (minore di $kSub$).

La sezione *controllo degli estremi* semplicemente impone il giusto ordine nella scelta degli estremi dei segmenti (*m1* dovrà avere numero di colonna e di riga inferiore rispetto a *m2*). La sezione *controllo lunghezza minima* controlla appunto che la lunghezza in parole dei segmenti (calcolabile come differenza tra le coordinate omonime -1) non sia inferiore al parametro $lMinSub$.

La nuova stored procedure *wordSubString()* è utilizzata dal DBMS per ottenere il segmento in questione a partire dalla frase completa.

3.9.3 I filtri

I filtri utilizzati hanno qui il compito di velocizzare l'analisi e la costruzione dei vari segmenti (scartando molte coppie a priori e riducendo così il calcolo della reciproca edit distance).

```
                                    -- filtro di lunghezza
AND ABS((m2.nc - m1.nc) - (m2.nr - m1.nr))
            <= ROUND((m2.nc - m1.nc +1)*<kSub>)

                                    -- filtro di conteggio
AND (SELECT COUNT(*)
     FROM MATCHPOS m1a
     WHERE m1a.cod1 = m1.cod1
     AND m1a.cod2 = m1.cod2
     AND m1a.nc <= m2.nc
     AND m1a.nc >= m1.nc
```

```
    AND m1a.nr <= m2.nr
    AND m1a.nr >= m1.nr
                                -- filtro di posizione
    AND ABS((m1a.nc − m1.nc)−(m1a.nr − m1.nr))
          <=ROUND((m2.nc − m1.nc +1)∗<kSub>)
) >= ALL ((m2.nc −m1.nc +1) − ROUND((m2.nc −m1.nc +1)∗<kSub>),
          (m2.nr −m1.nr +1) − ROUND((m2.nc −m1.nc +1)∗<kSub>))
```

Come si vede, si basano sulle stesse considerazioni e assunti spiegati nella sezione 3.6.1 per la query delle frasi intere e cui si rimanda per una spiegazione più dettagliata.

Ad esempio, il *filtro di lunghezza* impedisce che vengano ulteriormente analizzati segmenti di lunghezza molto diversa tra loro: ad esempio, un segmento di tre parole in una frase non potrà mai essere sufficientemente simile ad uno di dieci nell'altra. Allo stesso modo, vengono riadattati i *filtri di conteggio e posizione*: segmenti sufficientemente simili dovranno avere al loro interno un certo numero minimo di parole (dipendente dal tetto massimo di distanza richiesto, *kSub*), poste in posizioni non troppo differenti un segmento dall'altro.

Il filtro veramente nuovo in questo caso è il cosiddetto *filtro di inclusione*: questo si occupa di scartare tutti quei segmenti non ottimi, ad esempio quelli inclusi in altri. Da notare è che segmenti sovrapposti ma non inclusi uno nell'altro non vengono scartati, in quanto si riveleranno molto utili per "coprire" la frase cercata.

Questo filtro è ottenuto utilizzando una query innestata del tutto analoga come struttura a quella principale, in cui si verifica se non esistono segmenti (di estremi *m3* ed *m4*) aventi tutte le caratteristiche desiderate e che includano gli estremi di quello preso in esame nella query esterna (di estremi *m1* ed *m2*): se non ne esistono quest'ultimo sarà effettivamente uno dei segmenti cercati.

```
AND NOT EXISTS                  -- filtro di inclusione
    (                           -- parte analoga alla query esterna
     SELECT m3.nc, m3.nr, m4.nc, m4.nr
     FROM MATCHPOS m3, MATCHPOS m4, <tab1> r3, <tab2> r4
     WHERE m3.cod1 = m4.cod1
     AND m3.cod2 = m4.cod2
     AND m3.cod1 = r3.codice
```

```
AND m3.cod2 = r4.codice
AND m3.nr < m4.nr
AND m3.nc < m4.nc
AND (m4.nr − m3.nr + 1) >= <lMinSub>
AND ABS((m4.nc − m3.nc)−(m4.nr − m3.nr))
        <=ROUND((m4.nc − m3.nc +1)∗<kSub>)
AND (SELECT COUNT(∗)
        FROM MATCHPOS m3a
        WHERE m3a.cod1 = m3.cod1
        AND m3a.cod2 = m3.cod2
        AND m3a.nc <= m4.nc
        AND m3a.nc >= m3.nc
        AND m3a.nr <= m4.nr
        AND m3a.nr >= m3.nr
        AND ABS((m3a.nc − m3.nc)−(m3a.nr − m3.nr))
              <=ROUND(<kSub>∗(m4.nc−m3.nc +1))
    )>=ALL((m4.nc −m3.nc +1)−ROUND((m4.nc −m3.nc +1)∗<kSub>),
            (m4.nr −m3.nr +1)−ROUND((m4.nc −m3.nc +1)∗<kSub>))
AND wordEditDistanceDiag (wordSubString(r3.frase, m3.nr, m4.nr),
            wordSubString(r4.frase, m3.nc, m4.nc), <kSub>) >= 0
AND m3.cod2 = m1.cod2
                                    -- controllo di inclusione
AND ((m1.nc = m3.nc AND m4.nc > m2.nc)
      OR
      (m3.nc < m1.nc AND m2.nc = m4.nc)
      OR
      (m3.nc < m1.nc AND m4.nc > m2.nc)
      OR
      (m1.nc = m3.nc AND m2.nc = m4.nc
       AND m1.cod1 = m3.cod1 AND
         ((m1.nr = m3.nr AND m4.nr > m2.nr)
          OR
          (m3.nr < m1.nr AND m2.nr = m4.nr)
          OR
          (m3.nr < m1.nr AND m4.nr > m2.nr)
      ))))
```

Si ricorda che in appendice B.3 è riportata la query completa.

3.9.4 Un esempio esplicativo

Per chiarire meglio il compito della query, immaginiamo di aver impostato i parametri $lMinSub = 3$ e $kSub = 0.3$, che lo stemming sia attivato e che una delle frasi da ricercare sia (nelle due versioni originaria e con stemming):

23: So, welcome to the world of computer generated art.
(welcome world compute generate art)

Si supponga inoltre che nella Translation Memory siano presenti, tra le altre, le seguenti frasi:

2518: So, welcome to the world of music.
(welcome world music)
3945: We welcome our guests to the Madrid Art Expo!
(welcome guest Madrid art Expo)
5673: Welcome to the world of computer aided translation!
(welcome world compute aid translation)
10271: We welcome you to the world of computer generated fractals.
(welcome world compute generate fractal)
13456: This is a computer generated art work.
(be compute generate art work)

Tutte queste frasi presentano, nella versione elaborata, alcune parole in comune con la frase cercata, le cui "coordinate" saranno dunque state registrate nella tabella *MatchPos*. La query di ricerca delle sottoparti deve individuare le sottoparti migliori scegliendo opportunamente l'inizio e la fine del segmento simile ed applicando i vincoli richiesti.

Nell'esempio, la frase 2518 fornisce un solo possibile segmento: "welcome world". Poiché la sua lunghezza è di sole due parole in entrambe le frasi, esso non verrà considerato tra i possibili candidati ($2 < lMinSub = 3$).

La frase 3945 presenta due parole comuni; i segmenti candidati saranno pertanto quelli che nelle due frasi iniziano e finiscono con quelle parole: "welcome world compute generate art" e "welcome guest Madrid art". L'edit distance dei due segmenti è però elevata (3 su 5 parole cercate, $3/5 > kSub = 0.3$), pertanto anche questo candidato dovrà essere scartato.

Consideriamo infine le ultime tre frasi presentate: la frase 5673 presenta un segmento "welcome world compute". L'edit distance tra questo segmento e il

corrispondente della frase cercata (che risulterebbe nulla) non viene in questo caso nemmeno calcolata: la frase 10271 presenta un segmento "welcome world compute generate" che inizia nello stesso punto ma che copre una porzione maggiore della frase cercata (filtro di inclusione). Il segmento "compute generate art" della frase 13456 è invece considerato e analizzato in quanto copre una porzione della frase cercata (l'ultima parte) non inclusa in nessun'altra.

Pertanto, nella tabella dei risultati *SubMatch* verranno registrati i soli dati dei match "migliori", in questo caso quelli della frase 10271 e 13456 (vedi figura3.17). Come si vedrà nel capitolo seguente, al momento della presentazione di questi risultati, grazie agli algoritmi di allineamento, sarà possibile suggerire i segmenti delle corrispondenti traduzioni, che il traduttore potrà riutilizzare direttamente.

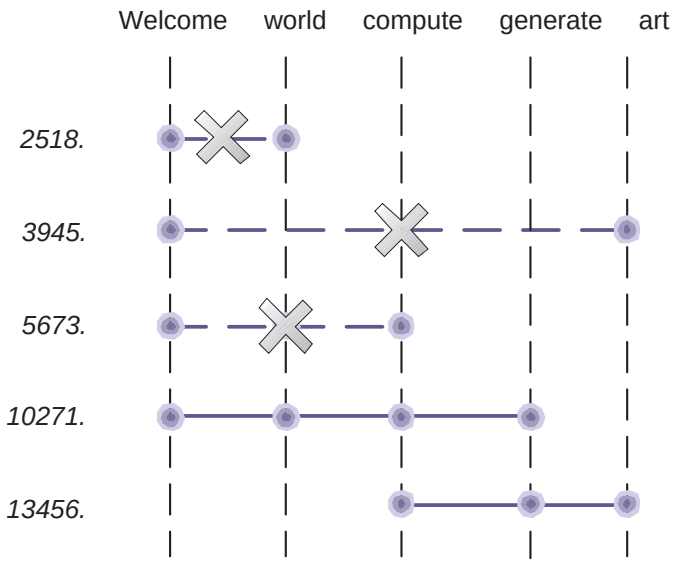


Figura 3.17: Ricerca delle sottoparti: un esempio

3.10 I parametri della ricerca: un riassunto

Ora che è stato descritto tutto il procedimento di ricerca, come ricapitolazione finale è opportuno riassumerne i vari parametri, che nel loro insieme forniscono una notevole flessibilità. Sono variabili a piacere:

- le dimensioni dei q-grammi (q e $qSub$);

- la massima distanza consentita tra frasi intere (k);
- la massima distanza consentita tra sottoparti di frase ($kSub$);
- la lunghezza minima in parole dei suggerimenti ($lMinSub$).

Inoltre è possibile attivare o meno, a seconda delle necessità:

- lo stemming delle frasi;
- la ricerca delle sottoparti simili e la relativa modalità (a un passo, a due passi veloce, a due passi accurata);
- i filtri delle query di ricerca (conteggio, posizione, lunghezza, e così via).

In questo modo il processo di ricerca è adattabile alle diverse esigenze dell'utente, e consente anche al ricercatore di indagare sul funzionamento delle varie componenti vedendo il comportamento nelle varie situazioni.

Capitolo 4

EXTRA: un ambiente EBMT

La ricerca di similarità descritta nel capitolo precedente è utilizzata, come si è visto, per sfruttare appieno il potenziale di una Translation Memory, fornendo al traduttore preziosi suggerimenti per la traduzione di nuovo materiale. Essa ha indubbiamente un'importanza fondamentale, ma per renderla pienamente utilizzabile e per avere un vero sistema sistema di Example-Based Machine Translation (EBMT) sono necessarie un certo numero di utilità e funzionalità aggiuntive.

Lo studio e il progetto di queste ulteriori funzionalità hanno portato alla creazione di un unico ambiente che fornisse al traduttore tutti i principali strumenti di cui necessitava, dal funzionamento integrato e richiamabili e configurabili comodamente tramite un'interfaccia comune. Questo ambiente è stato chiamato EXTRA (EXample-based TRanslation Assistant); in figura 4.1 è mostrata la finestra principale.

EXTRA non solo permette di pretradurre un documento utilizzando gli algoritmi di ricerca visti, ma fornisce anche una serie di strumenti che permettono di creare, gestire e analizzare una Translation Memory: partendo da semplici file di testo in svariati formati arriva ad inserire nel DB le varie frasi correttamente individuate, messe automaticamente in corrispondenza a quelle della lingua destinazione, elaborate ed allineate nelle due lingue anche internamente, a livello delle parole, per consentire di sfruttare tutta la potenza della ricerca di similarità parziale tra sottoparti.

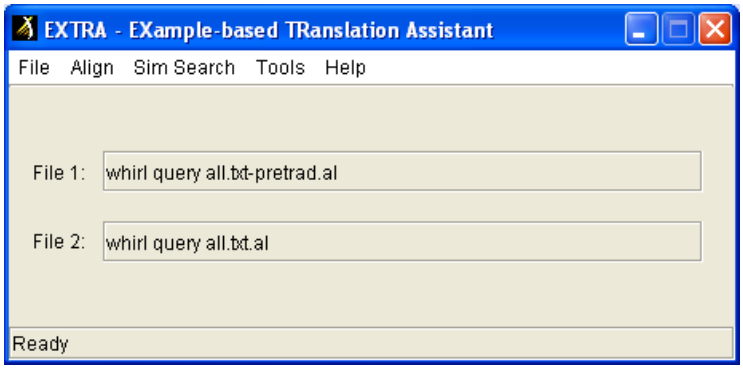


Figura 4.1: EXTRA: l'ambiente principale

4.1 Le funzionalità

Scendendo più in dettaglio, le principali funzionalità offerte (come si vede anche dai comandi disponibili nei menu, figura 4.2) sono le seguenti:

- Preparazione agli allineamenti
  (due versioni per altrettanti tipi di file di origine)
- Allineamento automatico delle frasi
- Allineamento automatico delle parole all'interno delle frasi
- Inserimento dei dati ottenuti nella Translation Memory
- Pretraduzione di un documento

Accanto a queste caratteristiche fondamentali, sono previsti anche:

- Strumenti per la gestione delle tabelle e l'analisi della Translation Memory (anche grafica, con la rappresentazione della distribuzione delle frasi presenti)
- Configurazione dei parametri di funzionamento

Nelle sezioni seguenti verranno analizzate tutte queste caratteristiche, con particolare attenzione a quei procedimenti che, anche in ambito di ricerca (vedi Capitolo 2), assumono una notevole importanza: gli allineamenti.

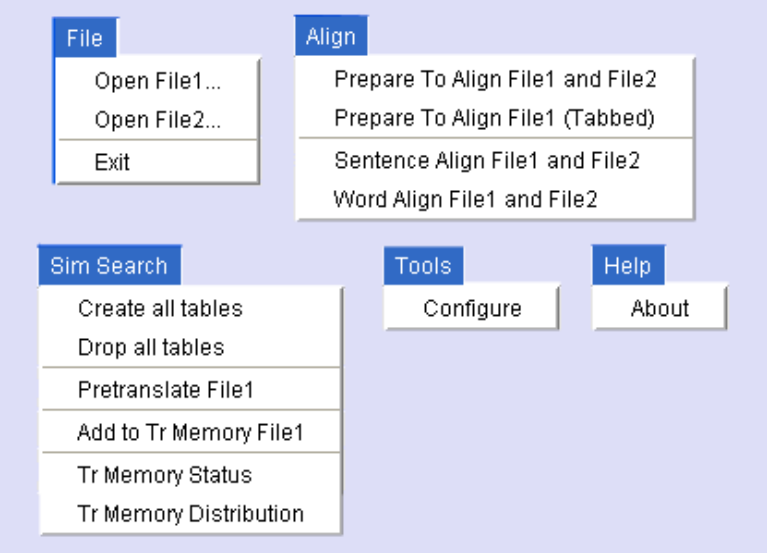


Figura 4.2: EXTRA: i comandi disponibili a menu

4.2 Preparazione agli allineamenti

Questa è l'operazione iniziale che viene effettuata sui file di testo contenenti i documenti (nelle due lingue) che si vorranno aggiungere alla Translation Memory: è necessaria per preparare questi documenti ai successivi processi di allineamento (figura 4.3).

Figura 4.3: Preparazione agli allineamenti: schema

Come si è visto non è infatti possibile inserire i documenti di testo così come sono inizialmente: nella Translation Memory dovranno essere inserite piuttosto le diverse *frasi* che li compongono, allineate nelle due lingue sia a livello delle frasi che a livello delle parole.

La preparazione dei file consiste fondamentalmente nella corretta individuazione dei paragrafi e delle frasi, la cui separazione verrà evidenziata esplicitamente nei file preparati prodotti in output: partendo da questi sarà possibile effettuare l'allineamento automatico delle frasi.

4.2.1 Il riconoscimento delle frasi

Per individuare nel modo più preciso e flessibile la corretta suddivisione in frasi di un file di testo, indispensabile poiché alla base di tutte le elaborazioni successive, è stato progettato un semplice automa a stati finiti (vedi figura 4.4). Gli stati finali rappresentano in questo caso i punti in cui vengono riconosciute la fine della frase e quella del paragrafo correnti.

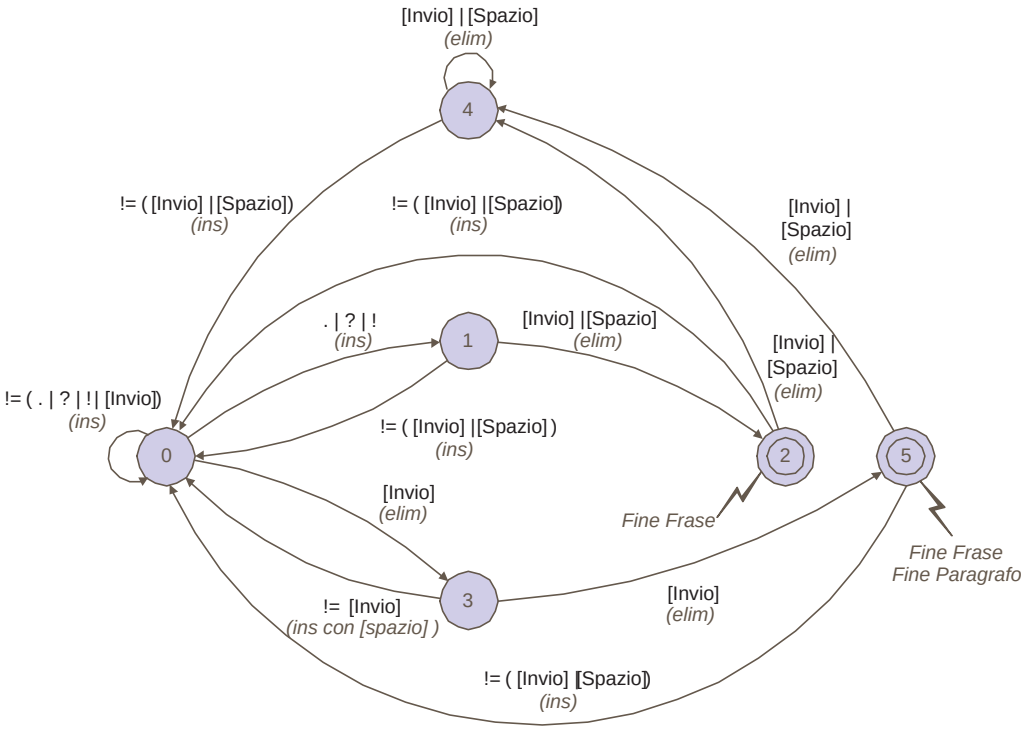


Figura 4.4: Riconoscimento delle frasi: l'automa

In particolare, la condizione di separazione tra una frase è l'altra è la presenza dei caratteri *'.'*, *'?'* o *'!'* seguiti da *[spazio]* o *[invio]*. Formalmente:

$('.' \mid '?' \mid '!')([spazio] \mid [invio])$.

La condizione di separazione tra un paragrafo e l'altro è invece un doppio *[invio]*.

Viene inoltre indicato se i caratteri di volta in volta riconosciuti vengono inseriti (*ins*) o eliminati (*elim*) nella frase estratta.

Con questo metodo si riescono a separare correttamente le frasi anche in documenti con una formattazione non ideale: ad esempio il carattere di fine linea non viene di per sé interpretato come la fine della frase (spesso è utilizzato in un file solo per non estendere la linea al di là di un certo numero di caratteri).

4.2.2 I file preparati

L'automa viene applicato ai due documenti (nelle due lingue), generando in output due file pronti per l'allineamento, di questo tipo:

```
Frase 1                    // paragrafo 1
.SENT                      // separatore di frasi
Frase 2                    // paragrafo 1
.SENT
.PARA                      // separatore di paragrafi
Frase 1                    // paragrafo 2
```

dove i separatori '`.SENT`' e '`.PARA`' sono modificabili dall'utente. In alternativa, è possibile preparare all'allineamento anche file contenenti frasi già correttamente segmentate ed individuate: sarà sufficiente rispondere negativamente alla richiesta di identificazione frasi (figura 4.5). In questo caso non verrà ovviamente applicato l'automa a stati finiti, ma le varie frasi (presenti già una per riga) saranno semplicemente portate in output aggiungendo gli opportuni separatori.

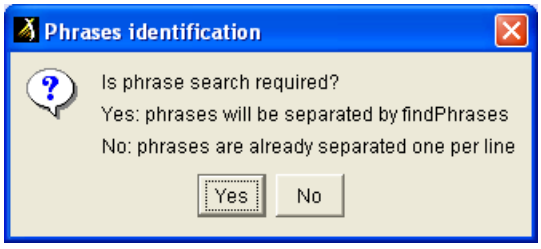


Figura 4.5: EXTRA: il requester per l'identificazione delle frasi

È anche prevista la gestione di documenti che, in un solo file, presentano non solo le frasi già separate ma anche già messe in corrispondenza a quelle della lingua destinazione, tipicamente in questo formato:

Frase 1 lingua sorgente	$< TAB >$	Frase 1 lingua destinazione
Frase 2 lingua sorgente	$< TAB >$	Frase 2 lingua destinazione
...		

In questo caso la preparazione riconduce i file a quello che sarebbe l'output dell'allineamento delle frasi (vedi sezione successiva) e permette di passare direttamente all'unico allineamento richiesto: quello delle parole.

4.3 Allineamento delle frasi

Dopo che i vari paragrafi e le rispettive frasi sono stati correttamente individuati e messi in evidenza nei file preparati, è possibile passare a un'operazione fondamentale nella creazione di una buona Translation Memory: l'allineamento delle frasi, cioè l'identificazione delle corrispondenze tra le frasi in una lingua e quelle nell'altra (la traduzione).

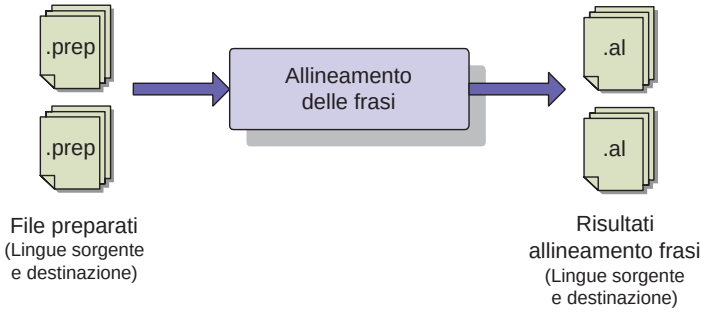


Figura 4.6: Allineamento frasi: schema

Si tratta di un problema abbastanza complesso poiché, come si può intuire, non è certo possibile confidare nel fatto che il traduttore abbia rispettato sempre tutti i limiti delle varie frasi: spesso, per rendere una traduzione più scorrevole, nella lingua destinazione può capitare di fondere due frasi di quella sorgente, o ancora di spezzare in due frasi un pensiero particolarmente lungo, o addirittura di sopprimere integralmente una frase (vedi figura 4.7).

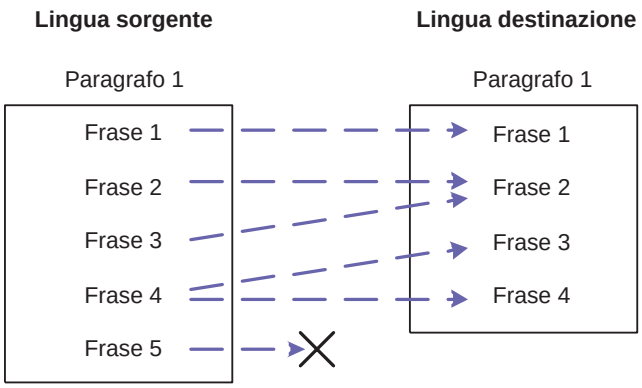


Figura 4.7: Allineamento frasi: esempi di corrispondenze

Proprio per questi motivi, l'allineamento è un processo piuttosto lungo e faticoso se svolto manualmente. In questo progetto viene utilizzato un metodo che *automaticamente* ricerca queste corrispondenze; esso è basato sui principi descritti da A. Gale e W. Church [12], nel loro lavoro già citato nel Capitolo 2.

4.3.1 I principi di funzionamento

Il metodo è basato su un modello statistico riguardante le lunghezze delle frasi (in caratteri). La considerazione fondamentale è la seguente: frasi più lunghe in una lingua *tendono* ad essere tradotte in frasi più lunghe nell'altra lingua, frasi più corte *tendono* ad essere tradotte in frasi più corte. Ad ogni possibile corrispondenza tra le frasi è assegnato un punteggio probabilistico che esprime una *distanza* tra esse, basata sulla differenza tra le lunghezze (in caratteri) tra le due frasi e sulla varianza di questa differenza. Questo punteggio probabilistico è quindi utilizzato in una struttura di programmazione dinamica per trovare il migliore allineamento delle frasi.

È sufficiente osservare la figura 4.8 per capire quanto questo approccio sia ben fondato: come si vede, le lunghezze (in caratteri) tra paragrafi, in questo caso inglesi e tedeschi, sono altamente correlate. L'asse orizzontale mostra la lunghezza dei paragrafi inglesi, la scala verticale le lunghezze dei corrispondenti paragrafi tedeschi.

Un grande punto di forza del metodo utilizzato è indubbiamente il fatto di essere comunque *completamente indipendente dalle lingue utilizzate* nei documenti: tutto è incentrato su modelli e calcoli matematici, garantendo in questo

modo non solo una applicabilità generalizzata ma anche una grande efficienza ed accuratezza.

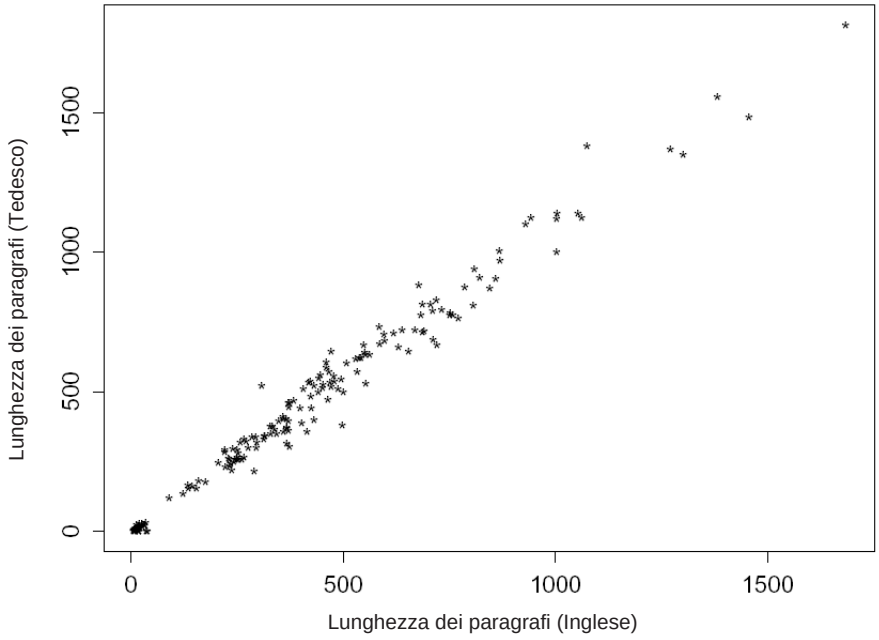


Figura 4.8: Allineamento frasi: correlazione nelle lunghezze dei paragrafi

4.3.2 La metrica di distanza

Come spiegato nel paragrafo precedente, alla base di tutto il calcolo di allineamento c'è una struttura di programmazione dinamica. La programmazione dinamica è spesso utilizzata per allineare due sequenze di simboli in moltissimi ambiti: genetica, riconoscimento del parlato, geologia, ecc. I dettagli delle tecniche di allineamento differiscono notevolmente da un'applicazione all'altra, ma tutti utilizzano una *metrica di distanza* per confrontare due singoli elementi delle sequenze, allo scopo di minimizzare le distanze totali tra gli elementi allineati all'interno delle due sequenze.

Per poter utilizzare questo approccio nell'allineamento delle frasi è dunque necessario innanzitutto definire una *distanza* adeguata a questo contesto.

La metrica di distanza utilizzata è basata su un modello probabilistico e in particolare sulla seguente assunzione: ogni carattere di testo in un linguaggio L_1

dà luogo ad un numero casuale di caratteri nell'altro linguaggio, L_2. Assumeremo che queste variabili aleatorie siano indipendenti e identicamente distribuite con una distribuzione normale.

Definizione 4.1 ***Il modello di distanza dell'allineamento.*** *Si considerino i due parametri fondamentali della distribuzione normale dei caratteri: c, la media, e s^2, la varianza. Con c si intenda il numero atteso di caratteri in L_2 per carattere in L_1, con s^2 la varianza del numero di caratteri in L_2 per carattere in L_1. Definiamo una nuova variabile aleatoria, δ, che mette in relazione le lunghezze delle due porzioni di testo considerate, l_1 ed l_2, presentando una distribuzione normale con media nulla e varianza unitaria:*

$$\delta = (l_2 - l_1 c)/\sqrt{l_1 s^2}$$

La misura di distanza cercata è data allora da una stima della probabilità condizionata $Prob(match|\delta)$, di cui si considererà il logaritmo per fare in modo che la somma di queste distanze produca i risultati desiderati:

$$-log(Prob(match|\delta)).$$

Con $Prob(match)$ si indica la probabilità legata ad un particolare tipo di corrispondenza nell'allineamento. I tipi considerati sono i seguenti:

- *1-1*: il caso più tipico, in cui ad una frase in una lignua ne corrisponde una ed una sola nell'altra;
- *1-0* o *0-1*: una frase in una lingua non ha corrispondenze nell'altra;
- *2-1* o *1-2*: ad una frase in una lingua ne corrispondono due nell'altra;
- *2-2*: a una coppia di frasi, considerata nel suo insieme, corrisponde una coppia di frasi nell'altra lingua.

Come dimostrato in [12], l'assunzione di distribuzione normale per δ è molto vicina alla realtà della maggior parte dei testi, soprattutto quando le due porzioni di testo in analisi sono effettivamente la traduzione l'una dell'altra. Il grafico della densità empirica presenta di solito un picco più pronunciato di una gaussiana, ma le differenze sono molto piccole e non influenzano l'algoritmo (figura 4.9).

Definita così la distanza, si pone ora il problema di come ottenerne una stima il più possibile precisa ed allo stesso tempo pratica da calcolare.

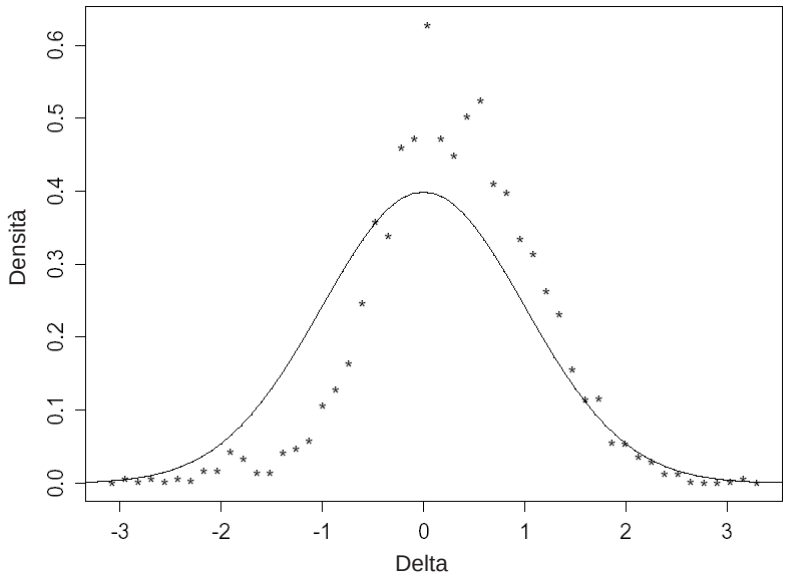


Figura 4.9: Allineamento frasi: la distribuzione di Delta

La stima della distanza

La quantità $-log(Prob(match|\delta))$ non è comodamente stimabile, ma con alcuni semplici passaggi matematici è possibile esprimerla in una maniera alternativa, che ne permetta un calcolo più diretto.

Utilizzando il teorema di Bayes, è possibile esprimere in questo modo la probabilità condizionata in questione:

$$Prob(match|\delta) = k \times Prob(\delta|match) \times Prob(match)$$

dove k è una costante che ai nostri fini può essere ignorata. La probabilità $Prob(\delta|match)$ può a sua volta essere espressa nel seguente modo:

$$Prob(\delta|match) = 2(1 - Prob(|\delta|)).$$

$Prob(|\delta|)$ è comodamente calcolabile in quanto, per come è stata scelta δ, corrisponde alla probabilità che una variabile aleatoria con una distribuzione normale, media nulla e varianza unitaria assuma valori maggiori di $|\delta|$. Questa è calcolabile con le formule note; in particolare, chiamata z questa variabile, risulta:

$$Prob(|\delta|) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{|\delta|} e^{-(z^2/2)} dz$$

Utilizzando questi passaggi e, per il calcolo di questo integrale, una approssimazione descritta da Abramowitz e Stegun [1], si ottiene pertanto la stima richiesta.

I valori dei parametri

L'unico punto rimasto ancora in sospeso circa il calcolo della distanza riguarda i valori da utilizzare per i parametri c ed s^2, nonché per la probabilità $Prob(match)$. Per tutti questi sono stati utilizzati di default i valori che Gale e Church [12] hanno determinato empiricamente dai numerosi test effettuati, permettendo comunque all'utente di variarli a seconda delle sue necessità e del tipo di testi utilizzati.

In particolare, c è stato stimato contando il numero di caratteri dei paragrafi di una lingua e dividendolo per il numero dei caratteri dei corrispondenti paragrafi nell'altra lingua. Per le lingue Inglese e Tedesco si è ottenuto $c \approx 1.1$, per Inglese e Francese $c \approx 1.06$; poiché la resa dell'algoritmo non è apparsa troppo sensibile a questi precisi valori, è stato scelto $c = 1$ come valore independente dalla lingua, adatto in particolare per le principali coppie di lingue europee.

Per quanto riguarda s^2, il modello assume che essa sia proporzionale alla lunghezza. La costante di proporzionalità è stata calcolata per le coppie Inglese-Tedesco ($s^2 \approx 7.3$) e Inglese-Francese ($s^2 \approx 5.6$). Anche in questo caso, queste differenze non sono troppo significative e viene scelto pertanto un più semplice valore indipendente dalle lingue, $s^2 = 6.8$.

Infine, per quanto riguarda la probabilità $Prob(match)$, legata come si è visto ai tipi di corrispondenze nell'allineamento, essa viene calcolata facendo riferimento ai valori determinati dagli allineamenti corretti di una serie di documenti di riferimento (vedi figura 4.10):

Categoria	**Prob(match)**
$1-1$	0.89
$1-0\ ;\ 0-1$	0.0099
$2-1\ ;\ 1-2$	0.089
$2-2$	0.011

Figura 4.10: Allineamento frasi: le categorie e le relative probabilità

4.3.3 L'algoritmo di programmazione dinamica

L'algoritmo di programmazione dinamica utilizzato ha il compito, per ciascuna coppia di paragrafi che formano i file forniti in input, di individuare il *miglior* allineamento possibile delle frasi al loro interno, quello cioè che minimizza la distanza descritta precedentemente.

Per ottenere questo, esso si basa sul calcolo ricorsivo; in particolare esso utilizza una funzione di distanza d che, attraverso i quattro parametri x_1, y_1, x_2 ed y_2, permette di esprimere i costi legati alle varie corrispondenze di allineamento viste. x ed y rappresentano, rispettivamente, le frasi del primo e del secondo documento da allineare.

$d(x_1, y_1; 0, 0)$	è il costo della *sostituzione* di x_1 con y_1
$d(x_1, 0; 0, 0)$	è il costo dell'*eliminazione* di x_1
$d(0, y_1; 0, 0)$	è il costo dell'*inserimento* di y_1
$d(x_1, y_1; x_2, 0)$	è il costo della *contrazione* di x_1 ed x_2 in y_1
$d(x_1, y_1; 0, y_2)$	è il costo dell'*espansione* di x_1 in y_1 ed y_2
$d(x_1, y_1; x_2, y_2)$	è il costo della *fusione* di x_1 ed x_2, sostituiti da $y_1$ed y_2

Passando a questa funzione le lunghezze delle frasi in questione, essa restituisce un valore di distanza calcolato come spiegato nei paragrafi precedenti. L'equazione di ricorsione vera e propria è allora definibile formalmente come segue:

Definizione 4.2 ***L'equazione di ricorsione.*** *Si indichino con s_i, i=1...I, le frasi in una lingua e con t_j, j=1...J, le traduzioni di queste frasi nell'altra lingua. Sia d la funzione di distanza descritta e D(i,j) la minima distanza tra le frasi $s_1, ..., s_i$ e le loro traduzioni $t_1, ..., t_j$, nel caso di allineamento migliore. D(i,j) è calcolata minimizzandone il calcolo sui sei casi possibili (sostituzione, cancellazione, inserimento, contrazione, espansione e fusione): partendo dalla condizione iniziale D(i,j)=0, D(i,j) è definita dalla seguente ricorsione:*

$$D(i,j) = min \begin{cases} D(i, j-1) & + & d(0, t_j; 0, 0) \\ D(i-1, j) & + & d(s_i, 0; 0, 0) \\ D(i-1, j-1) & + & d(s_i, t_j; 0, 0) \\ D(i-1, j-2) & + & d(s_i, t_j; 0, t_{j-1}) \\ D(i-2, j-1) & + & d(s_i, t_j; s_{i-1}, 0) \\ D(i-2, j-2) & + & d(s_i, t_j; s_{i-1}, t_{j-1}) \end{cases}$$

Utilizzando la ricorsione indicata, per ogni paragrafo viene compilata una matrice di distanza (D) che, unita a due matrici ausiliarie ($PathX$ e $PathY$) per tenere traccia del "percorso" di allineamento seguito, permette di risalire all'allineamento più probabile, quello migliore per le frasi del paragrafo preso in considerazione.

4.3.4 Un esempio del calcolo

Per chiarire con un esempio il calcolo delle tre matrici utilizzate per l'allineamento tra frasi (D, $PathX$ e $PathY$), immaginiamo un caso molto semplice: due documenti contenenti un solo paragrafo ciascuno, formato da due frasi.

Documento X: paragrafo 1: frase 1: lunghezza 15
frase 2: lunghezza 5
Documento Y: paragrafo 1: frase 1: lunghezza 10
frase 2: lunghezza 6

L'allineamento, utilizzando la ricorsione vista, procede calcolando da sinistra verso destra una riga dopo l'altra i vari elementi delle matrici D, $PathX$ e $PathY$. Ciascuna di queste matrici avrà tante righe quante sono le frasi del documento X (più una iniziale) e tante colonne quante sono le frasi del documento Y (più una iniziale).

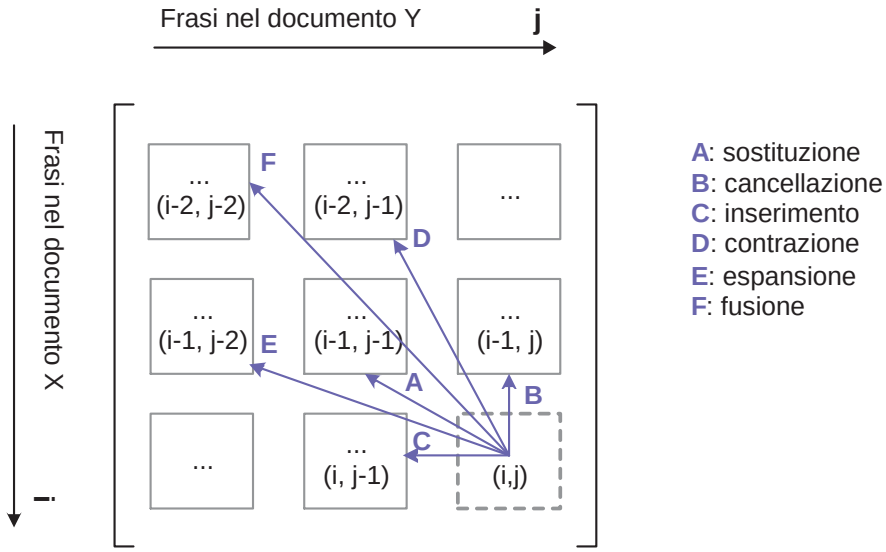


Figura 4.11: Allineamento frasi: gli elementi interessati dal calcolo di (i,j)

Ognuno degli elementi della matrice D viene calcolato osservando i valori degli elementi calcolati precedentemente (figura 4.11) sommati ai costi corrispondenti ai relativi "salti": ad esempio il valore dell'elemento (i, j) sarà la minore distanza tra quelle risultanti dall'elemento $(i-1, j-1)$ più il relativo costo di sostituzione, dell'elemento $(i, j-1)$ più il relativo costo di inserimento, e così via. Come si vede, il valore di ogni cella (i, j) della matrice D contiene la minima distanza totale corrispondente al miglior allineamento delle frasi fino a quel punto.

In ciascuna cella delle matrici $PathX$ e $PathY$ vengono invece registrate le coordinate della cella da cui si proviene nel percorso di allineamento seguito.

Per le frasi dell'esempio, le matrici risultano le seguenti:

$$D = \begin{bmatrix} 0 & 694 & 1313 \\ 783 & 53 & 238 \\ 1382 & 343 & 66 \end{bmatrix};\ PathX = \begin{bmatrix} - & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix};\ PathY = \begin{bmatrix} - & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Esaminando le matrici $PathX$ e $PathY$ partendo dalla cella in basso a destra, è possibile ricostruire a ritroso il miglior allineamento trovato: dalla cella $(2, 2)$ si passa alla cella $(1, 1)$ quindi alla cella di partenza $(0, 0)$ (figura 4.12).

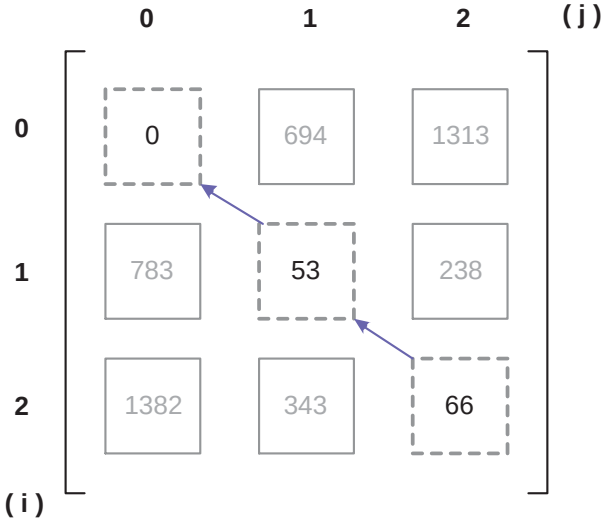


Figura 4.12: Allineamento frasi: il percorso di allineamento

Ognuno di questi "movimenti" corrisponde ad un allineamento, la cui distanza si calcola sottraendo il valore della cella di partenza di D a quella di arrivo:

$(2,2) \rightarrow (1,1) \Rightarrow$ (sostituzione) Distanza = 66-53 = 13 (frase 2)
$(1,1) \rightarrow (0,0) \Rightarrow$ (sostituzione) Distanza = 53-0 = 53 (frase 1)

L'allineamento risultante è allora (nel formato (X_1, Y_1, X_2, Y_2, D)):

15 10 0 0 53 (frase lunga 15 allineata con frase lunga 10, distanza 53)
5 6 0 0 13 (frase lunga 5 allineata con frase lunga 6, distanza 13)

La distanza totale del miglior allineamento trovato per questo paragrafo è pertanto 66 (53+13).

Il risultato dell'allineamento di questo esempio è abbastanza scontato ma è stato volutamente reso molto semplice per non complicare eccessivamente la trattazione e far comunque capire i passi fondamentali del calcolo dinamico. Esempi più significativi dal punto di vista dei risultati ottenuti saranno mostrati nel Capitolo 6.

4.3.5 I parametri disponibili: un riassunto

Si riassumono qui di seguito i parametri disponibili per l'allineamento delle frasi, per fornirne una visione unitaria. Sono variabili dall'utente:

- i delimitatori di frase ($softDelimiter$) e paragrafo ($hardDelimiter$), nonché quello di distanza ($distDelimiter$)
- la media della distribuzione normale dei caratteri (c)
- la varianza della distribuzione normale dei caratteri ($s2$)

4.3.6 I file risultanti

Il risultato dell'allineamento delle frasi sono due file (uno per lingua) con estensione *.al*, che presentano la seguente struttura:

```
Frase 1
.DIST xx                    // indicatore di distanza
.SENT                       // separatore di frasi
Frase 2
.DIST xx
.SENT
```

Come si vede, la separazione tra paragrafi, utilizzata per l'allineamento delle frasi, non è più necessaria e pertanto non compaiono più i separatori di paragrafo. Compare invece l'indicatore di distanza seguito dalla distanza di allineamento di ciascuna coppia di frasi (essa verrà poi inserita insieme agli altri dati nella Translation Memory).

4.4 Allineamento delle parole

In EXTRA, l'allineamento delle frasi è solo un primo passo, un passo indispensabile ma insufficiente a sfruttare appieno il potenziale della Translation Memory. Come si è visto nel capitolo precedente, uno dei punti di forza di questo ambiente è il fatto che la ricerca di somiglianza non si limiti ad analizzare le frasi intere ma consideri anche eventuali sottoparti.

Ovviamente, la ricerca avviene sulle frasi nella lingua sorgente: la nuova frase da tradurre viene confrontata con le frasi precedentemente tradotte. Quello che però interessa al traduttore e che il programma deve suggerire non è tanto la frase simile a quella da tradurre ma la *traduzione* di quest'ultima. È proprio qui che l'allineamento delle parole si rivela indispensabile: le frasi della translation memory e le relative traduzioni vengono analizzate, mettendo in corrispondenza le varie parole una con l'altra. Questi dati di allineamento vengono quindi registrati nella Translation Memory, per poi essere sfruttati al momento dell'estrazione dei suggerimenti: grazie ad essi è possibile sapere a quale parte di frase tradotta (che verrà suggerita) corrisponde la parte in lingua sorgente riconosciuta come simile.

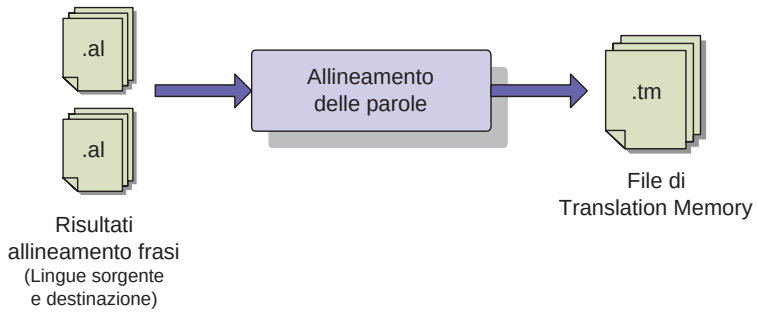


Figura 4.13: Allineamento parole: schema

Partendo allora dai file risultanti dall'allineamento delle frasi, l'ulteriore pas-

saggio per arrivare alla costruzione della Translation Memory completa di tutti i dati richiesti è l'allineamento delle parole: in figura 4.13 ne è mostrato lo schema.

4.4.1 I principi di funzionamento

Il problema che l'allineamento delle parole deve risolvere è il seguente: date una frase in lingua sorgente e la corrispondente frase in lingua destinazione, è necessario trovare la *funzione di corrispondenza* tra esse che più possibile rispecchi la realtà.

Nello spazio bitestuale formato dalle due frasi (si veda anche la sezione 2.3.1) è possibile rappresentare graficamente tale funzione di corrispondenza tra le parole: i valori sull'asse delle ascisse rappresentano le parole della frase da tradurre, quelli sull'asse delle ordinate le parole della corrispondente traduzione. Se le due frasi fossero esattamente identiche (corrispondenza perfetta) la funzione coinciderebbe con la diagonale principale dello spazio bitestuale (la linea retta che lo attraversa dall'angolo in basso a sinistra all'angolo in alto a destra); poiché a seconda delle lingue usate l'ordine e il numero delle parole può subire variazioni, la funzione di allineamento effettiva sarà una linea spezzata che "oscilla" intorno alla diagonale. In figura 4.14 è mostrato un esempio di questa funzione per due frasi di esempio (in questo caso per una traduzione da inglese a italiano).

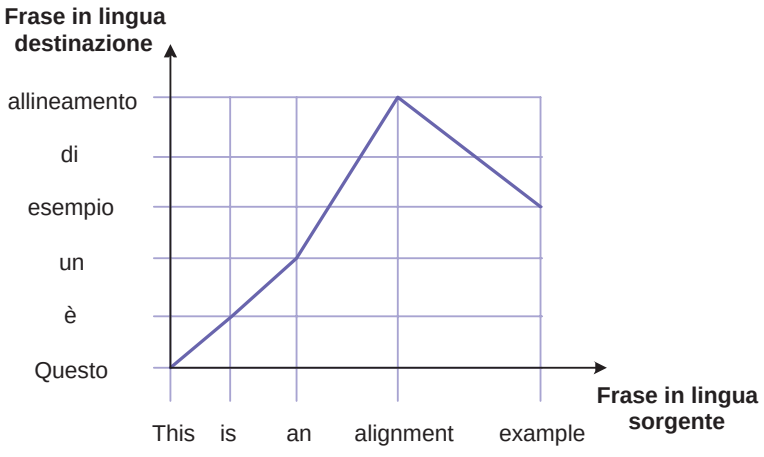


Figura 4.14: Allineamento parole: un esempio

Come si intuisce dalla figura, la funzione ha il seguente significato: data una "posizione" della frase da tradurre essa restituisce la "posizione" corrispondente

nella frase tradotta. Ad esempio, alla parola “alignment” la funzione mette in relazione la parola “allineamento”.

Ovviamente non sempre è possibile tracciare questa linea di corrispondenza in modo del tutto univoco ed esatto; il compito di questo allineamento non è d’altronde quello di stabilire con certezza a quale precisa parola corrisponda ogni singola parola della frase da tradurre. Ciò che si vuole è invece stabilire con buona approssimazione a quale segmento della frase destinazione corrisponda un dato segmento della frase sorgente.

Sempre osservando l’esempio in figura si capisce che non tutte le parole di una lingua sono in precisa relazione con quelle dell’altra: ad esempio la parola “di” in italiano non corrisponde a nessuna precisa parola inglese. Tuttavia, osservando il grafico, si capisce bene che se siamo interessati ad esempio alla traduzione del segmento “alignment example”, esso è in chiara corrispondenza con le parole “esempio di allineamento”, che è proprio il suggerimento che il programma potrà fornire. Come si vede, il compito dell’algoritmo di allineamento sarà pertanto quello di trovare, nel modo che si vedrà, solo alcuni punti di corrispondenza, quelli più certamente individuabili (in questo caso “example” ed “esempio”, “alignment” ed “allineamento”), e dall’unione (interpolazione) di questi punti base risalire all’allineamento completo.

4.4.2 Schema del procedimento

Il procedimento seguito per realizzare l’allineamento delle parole è stato concepito partendo dalle idee migliori presenti nella ricerca di questo campo (ad esempio i lavori di Simard [38] o di Church [6]) e producendo un algoritmo che combinasse i pregi dei vari approcci, proponendo allo stesso tempo diverse idee innovative. Le caratteristiche fondamentali sono:

- identificazione ed allineamento delle parole (quali sigle, nomi propri, ecc.) rimaste immutate nelle due lingue;
- identificazione ed allineamento della punteggiatura;
- identificazione ed allineamento delle cosiddette *parole affini* (chiamate in inglese *cognate*) attraverso un algoritmo di ricerca di somiglianza;
- allineamento basato su *punteggio*: un elemento (parola, punteggiatura) della lingua sorgente è messo in relazione (se possibile) a quello della lingua

destinazione con cui ha un maggiore punteggio di affinità (dipendente non solo dalle parole in sé ma anche dalla posizione che esse assumono nelle frasi);

- notevoli *efficienza* e *precisione*;
- *indipendenza dalla coppia di lingue* analizzate;
- *flessibilità*: grazie ai numerosi parametri; previsti, l'algoritmo risulta adattabile alla maggior parte delle esigenze.

Il procedimento adottato prevede, per ogni coppia di frasi, i passi mostrati in figura 4.15:

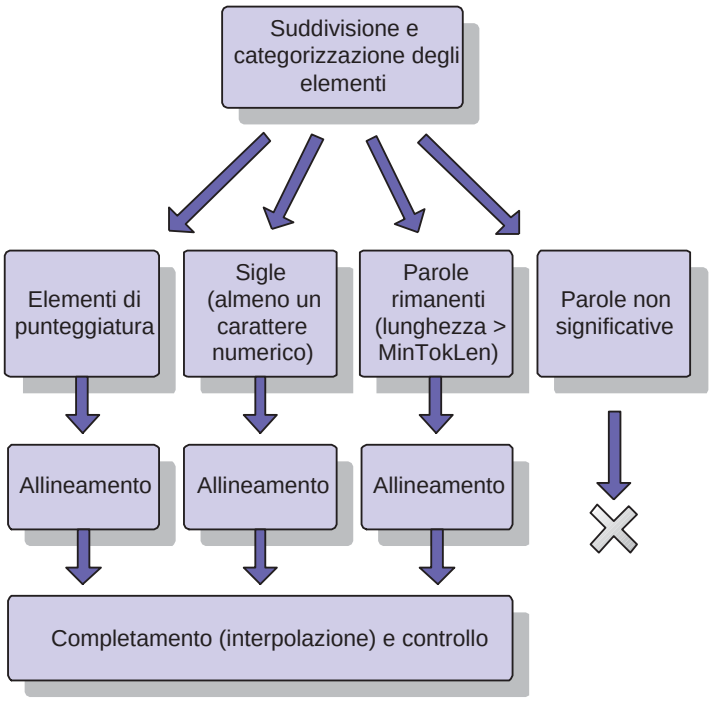


Figura 4.15: Allineamento parole: funzionamento

1. scomposizione delle frasi in *elementi (token)*;
2. suddivisione di tali elementi in categorie;
3. ricerca del migliore allineamento per ciascun elemento di queste categorie;

4. controllo ed interpolazione dell'allineamento per ottenere l'allineamento finale.

Le categorie in cui vengono suddivisi gli elementi sono:

- elementi di punteggiatura (punti, virgole, parentesi, virgolette e così via);
- sigle e termini alfanumerici (parole contenenti almeno un carattere numerico);
- parole non contenenti caratteri numerici e di lunghezza maggiore o uguale ad un parametro configurabile (*minTokLen*, normalmente impostato a 4 caratteri).

Per ciascuna di queste categorie viene eseguito separatamente l'algoritmo di allineamento: questo è possibile grazie alla disgiunzione tra i loro elementi (elementi di punteggiatura di una frase potranno allinearsi solo con gli elementi di punteggiatura dell'altra, e così via). La presenza del parametro *minTokLen* permette di scartare dall'allineamento quelle parole poco significative e tra le quali sarebbe difficile se non a volte impossibile trovare una corrispondenza: l'allineamento di queste risulterà comunque dall'interpolazione finale ed il processo risulta così più veloce ed affidabile.

4.4.3 L'algoritmo di allineamento

L'algoritmo prevede l'utilizzo dei seguenti array numerici, contenenti ciascuno tanti elementi quante sono le parole della frase della lingua sorgente:

- *Pos[]*: posizione dell'elemento della frase destinazione cui l'elemento della frase sorgente è associato
- *Score[]*: punteggio dell'associazione tra gli elementi

Ad esempio, l'assegnazione $Pos[3] = 4$ equivale ad associare al terzo elemento della lingua sorgente il quarto della lingua destinazione. Utilizzando queste strutture dati viene mantenuto il carattere della funzione di allineamento: poiché ad ogni elemento corrisponde una specifica posizione nell'array, ad ogni elemento della frase sorgente non corrisponderà mai più di un elemento della relativa traduzione.

L'algoritmo ha la seguente struttura:

```
∀ categoria di elementi
    ∀ elemento j della frase destinazione
        bestScore = minScore
        bestPos = -1

        // ricerca del miglior allineamento per l'elemento corrente
        ∀ elemento i della frase sorgente
            Calcola il punteggio di affinità score(i,j)
            Se score(i,j) > bestScore
                bestScore = Score(i,j)
                bestPos = i

        Se bestPos != -1    // trovato un possibile allineamento
            pos[bestPos]=j
            score[bestPos]=bestScore
```

Gli elementi della frase destinazione vengono esaminati uno dopo l'altro e per ciascuno viene ricercato il miglior accoppiamento possibile con gli elementi della frase sorgente; alla fine del processo avremo per ciascun elemento della frase sorgente il migliore accoppiamento (se trovato).

È previsto anche un parametro *minScore*, che come si vede funge da soglia minima: gli accoppiamenti con punteggio inferiore a quello specificato non saranno presi in considerazione.

4.4.4 Il punteggio di affinità

Tutto l'algoritmo si basa sui punteggi assegnati alle varie coppie di parole: più questo punteggio è alto più è l'accoppiamento è "forte", cioè più è probabile che le parole siano effettivamente in corrispondenza.

Il sistema di punteggi utilizzato è strutturato nella maniera più possibile flessibile e prevede l'assegnazione di un punteggio non solo alle parole riconosciute come identiche, ma anche, seppur in modo inferiore, a quelle riconosciute come *simili*. Inoltre, a parità di somiglianza, vengono premiate le parole che figurano in posizioni più simili nelle due frasi: più la coppia di parole si trova distante nelle due frasi (ad esempio una parola all'inizio di una frase ed una alla fine

dell'altra) più il relativo punteggio risulterà diminuito da una opportuna *funzione di smorzamento*.

Uguaglianza

Il punteggio di uguaglianza è il massimo ottenibile: due termini uguali ottengono, a meno dello smorzamento, un punteggio quantificato dal parametro $eqMult$ (impostato di default a 1500). Si tratta del punteggio più elevato poiché due termini uguali nelle due lingue rappresentano un punto di corrispondenza praticamente sicuro, ed è pertanto giusto che pesino di più, ad esempio, di due termini solamente simili.

$$score_{eq}(A, B) = eqMult$$

Il punteggio di uguaglianza è assegnabile a tutte le classi di termini allineabili; in particolare, per gli elementi di punteggiatura e le sigle alfanumeriche, il test di uguaglianza è il solo preso in considerazione (per essi non avrebbe senso parlare di somiglianza).

Somiglianza ed LCS

Per la sola categoria delle parole non alfanumeriche e di lunghezza superiore alla soglia minima, oltre al controllo di uguaglianza, in caso di esito negativo dello stesso, viene applicato un algoritmo che quantifica la somiglianza tra le due parole in questione. Grazie a questo algoritmo è possibile riconoscere un buon numero di ulteriori accoppiamenti, quelli riguardanti le cosiddette parole *affini*: parole con una morfologia molto simile nelle due lingue. Per fare un esempio: le parole "government" (inglese) e "gouvernement" (francese) non sono del tutto identiche ma, anche non conoscendo le due lingue, è facile intuirne la stretta relazione.

Ricercare queste somiglianze di forma è fruttuoso per la maggior parte delle lingue e soprattutto è economico: non sono richieste informazioni aggiuntive quali dizionari e thesauri. In ambito di ricerca sono stati proposti diversi metodi per la ricerca di queste somiglianze (vedi sezione 2.4.1): ricercare parole che iniziano con un certo numero di caratteri uguali, o a che abbiano un certo numero di caratteri uguali consecutivi.

L'approccio adottato in EXTRA va oltre tutto questo, permettendo di trovare gli accoppiamenti in modo più completo ed efficace. Si utilizza un algoritmo

chiamato *LCS (Longest Common Subsequence)*, molto simile a quello di edit distance e considerabile quasi come il duale di quest'ultimo: anche qui viene calcolata dinamicamente una matrice, ma i relativi valori invece di essere distanze sono lunghezze. Più il risultato finale è alto più le due parole presentano affinità morfologiche. In questo caso le varie colonne e righe della matrice corrispondono ai caratteri delle varie parole. La definizione formale classica è la seguente:

Definizione 4.3 ***Common Subsequence.*** *Siano $A = a_1a_2...a_m$, $B = b_1b_2...b_n$ due sequenze (parole) di un alfabeto Σ di dimensione s. Una sequenza $S = s_1s_2...s_l$, $l \leq m$, $l \leq n$, è detta sequenza comune (Common Subsequence) di A e B sse S può essere ottenuta da A e B cancellando un certo numero di simboli, non necessariamente consecutivi.*

Per calcolare la LCS tra due parole si utilizza un algoritmo dinamico che prevede la compilazione della matrice L: definendo $|A|$ la lunghezza della parola A e A_i, $0 \leq i \leq |A|$, il prefisso di A di lunghezza i, l'elemento $L_{i,j}$ è così definito:

$L_{i,j} = max\{|S| : \text{S e' una sequenza comune di } A_i \text{ e } B_j\}$

L'equazione di ricorsione è la seguente (descritta in [34]):

$$L_{i,j} = \begin{cases} 0 & se \quad i = 0 \; o \; j = 0 \\ L_{i-1,j-1} + 1 & se \quad a_i = b_i \\ max\{L_{i-1,j}, L_{i,j-1}\} & se \quad a_i \neq b_j \end{cases}$$

Pertanto per calcolare il punteggio di LCS tra le parole "example" ed "esempio" si costruirà la matrice mostrata in figura 4.16: il risultato ottenuto è una lunghezza di tre caratteri (cella in basso a destra).

È opportuno inoltre rapportatare questo risultato assoluto alla lunghezza delle parole confrontate: dividendo il valore ottenuto dal calcolo per il massimo delle lunghezze delle due parole si ottiene un valore compreso tra 0 (parole diverse) e 1 (parole identiche). Verranno considerate simili le parole con un punteggio superiore al parametro $minLCS$, normalmente impostato a 0.4: per l'esempio il punteggio è 3/7=0.43> $minLCS$, pertanto le due parole vengono considerate (giustamente) affini e messe in corrispondenza.

Il punteggio di somiglianza risulta:

$$score_{sim}(A, B) = lcsMult \times \frac{LCS}{max\{|A|, |B|\}}$$

		E	X	A	M	P	L	E
	0	0	0	0	0	0	0	0
E	0	1	1	1	1	1	1	1
S	0	1	1	1	1	1	1	1
E	0	1	1	1	1	1	1	2
M	0	1	1	1	2	2	2	2
P	0	1	1	1	2	3	3	3
I	0	1	1	1	2	3	3	3
O	0	1	1	1	2	3	3	**3**

Figura 4.16: Allineamento parole: LCS, un esempio di calcolo

Esso è calcolato moltiplicando il valore relativo dell'LCS per il moltiplicatore *lcsMult*; essendo *lcsMult* impostato di default a 1000, i punteggi di affinità delle parole saranno in questo caso compresi tra 0 e 1000.

La funzione di smorzamento

I valori di punteggio visti per uguaglianza e somiglianza vengono poi smorzati da un'opportuna funzione di smorzamento, che risente della distanza relativa tra le posizioni nelle rispettive frasi delle due parole confrontate. È evidente, per esempio, che se le frasi in questione contengono due virgole, una in posizione iniziale ed una in posizione finale, è più probabile che la prima virgola incontrata in una frase sia da allineare con la prima incontrata nell'altra, e così via: per ottenere questo comportamento è necessario che il punteggio tra elementi in posizioni vicine, a parità di uguaglianza o somiglianza, sia maggiore di quello in posizioni lontane.

La funzione di smorzamento è così definita:

$$s(p_A - p_B) = minDecay + \frac{(1 - minDecay)}{(1 + coeff \times (p_A - p_B)^2)}$$

In figura 4.17 ne è mostrato l'andamento per $minDecay = coeff$=0.2 (i valori standard).

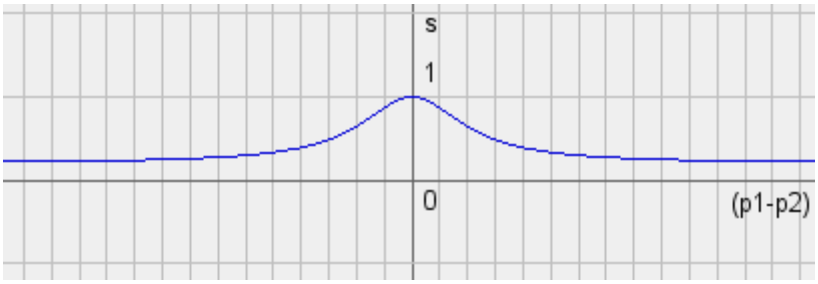


Figura 4.17: Allineamento parole: la funzione di smorzamento

Il punteggio finale si ottiene moltiplicando il punteggio di uguaglianza o somiglianza per il valore di questa funzione:

$$score(A, B) = score_{eq/sim}(A, B) \times s(p_A - p_B)$$

Ovviamente p_A e p_B rappresentano le posizioni delle parole all'interno delle frasi: quando le due parole si trovano nella stessa posizione lo smorzamento risulterà nullo ($s(0) = 1$), più la posizione sarà differente più lo smorzamento sarà elevato (tendente asintoticamente a $minDecay = 0.2$).

C'è da notare un ultimo aspetto di questa funzione: le posizioni delle parole p_A e p_B, prima di essere utilizzate nel calcolo, vengono rese confrontabili. Ad esempio se la frase in cui è presente la parola A è di 20 parole mentre la frase in cui è presente la parola B è di 10, la posizione p_B verrà prima moltiplicata per 2 (il rapporto tra le lunghezze):

$$p_B = p_B \times \frac{len(FraseA)}{len(FraseB)}$$

In questo modo, ad esempio, la ventesima parola della prima frase e la decima parola della seconda ottengono punteggio pieno, poiché in posizioni relative identiche (le ultime delle rispettive frasi).

4.4.5 Interpolazione e verifica finali

Ottenuta la migliore serie di punti di allineamento, come passo finale è necessario congiungerli ed interpolarne i valori per i termini intermedi.

Innanzitutto si provvede ad aggiungere i punti iniziali e finali, se non presenti; quindi, per ogni coppia di punti, si immagina di tracciare un segmento che li congiunge e si assegnano i valori compresi tra gli estremi alle parole intermedie.

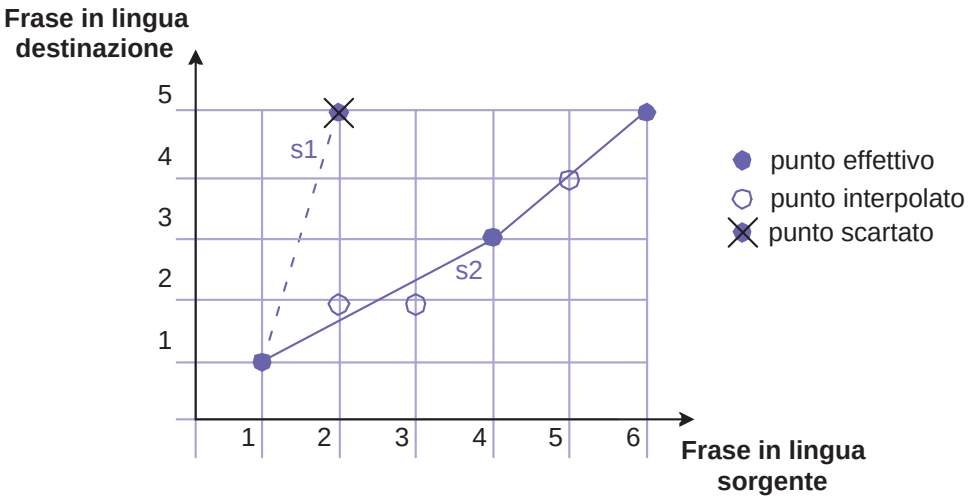


Figura 4.18: Allineamento parole: interpolazione e verifica, un esempio

Nell'esempio di figura 4.18 le parole della frase sorgente 2, 3 e 5 hanno ottenuto dall'interpolazione i propri valori di allineamento.

In questa fase finale è previsa anche una verifica dei punti trovati, per recuperare alcuni possibili errori delle fasi precedenti. La verifica consiste nell'evitare di creare una funzione di allineamento con picchi troppo pronunciati: nell'esempio, il punto della parola 2 della frase sorgente avrebbe creato un picco eccessivo (pertanto poco probabile) nell'allineamento e non viene pertanto considerato. Per la precisione, il test effettuato è il seguente: dato un punto di partenza, vengono considerati i segmenti s_1 (tratteggiato in figura) ed s_2 che lo congiungono ai due punti che lo seguono, se presenti. Il segmento s_1 (ed il suo punto finale) viene scartato se valgono entrambe queste condizioni:

- il coefficiente angolare di s_1 è maggiore (in modulo) di *maxSlRatio* volte il coefficiente angolare di s_2;

- il punteggio relativo al punto finale di s_2 è maggiore di *maxScRatio* volte il punteggio del punto finale di s_1.

Vengono scartati insomma i punti con basso punteggio e che creerebbero picchi anomali nella funzione di allineamento.

Si ricorda che nel Capitolo 6, dedicato ai risultati ottenuti, saranno mostrati esempi di funzionamento su alcune frasi significative.

4.4.6 I parametri disponibili: un riassunto

Come al solito, riassumiamo qui di seguito i parametri disponibili per il procedimento visto, in questo caso l'allineamento delle parole, in modo da darne un quadro unitario. Sono modificabili a seconda delle esigenze:

- la minima lunghezza di una parola allineabile ($minTokLen$);
- il punteggio di allineamento minimo richiesto ($minScore$);
- la percentuale di somiglianza minima richiesta ($minLCS$);
- i punteggi massimi di uguaglianza ($eqMult$) e somiglianza ($lcsMult$);
- i parametri della funzione di smorzamento ($coeff$ e $minDecay$);
- i parametri di verifica e correzione dell'allineamento finale ($maxSlRatio$ e $maxScRatio$).

4.4.7 Il file di Translation Memory

Il risultato del processo di allineamento delle parole è un file unico, di estensione *.tm*, contenente finalmente tutti i dati richiesti per l'inserimento delle frasi nella Translation Memory (a meno delle elaborazioni finali, quali lo stemming). Il formato è il seguente:

```
Frase 1                // lingua sorgente
Frase 1                // lingua destinazione
<Posizioni e punteggi di allineamento delle parole>
<Distanza di allineamento delle frasi>

Frase 2                // lingua sorgente
...
```

La distanza di allineamento delle frasi deriva dal precedente passo di allineamento, quello delle frasi.

4.5 Aggiunta di dati alla TM

Dopo aver preparato ed allineato i file di testo, si è ottenuto un file *.tm* contenente le coppie di frasi accompagnate dalle informazioni di allineamento: a questo punto si dispone di tutti i dati per effettuare l'inserimento nella Translation Memory (la figura 4.19 riepiloga tutto il procedimento).

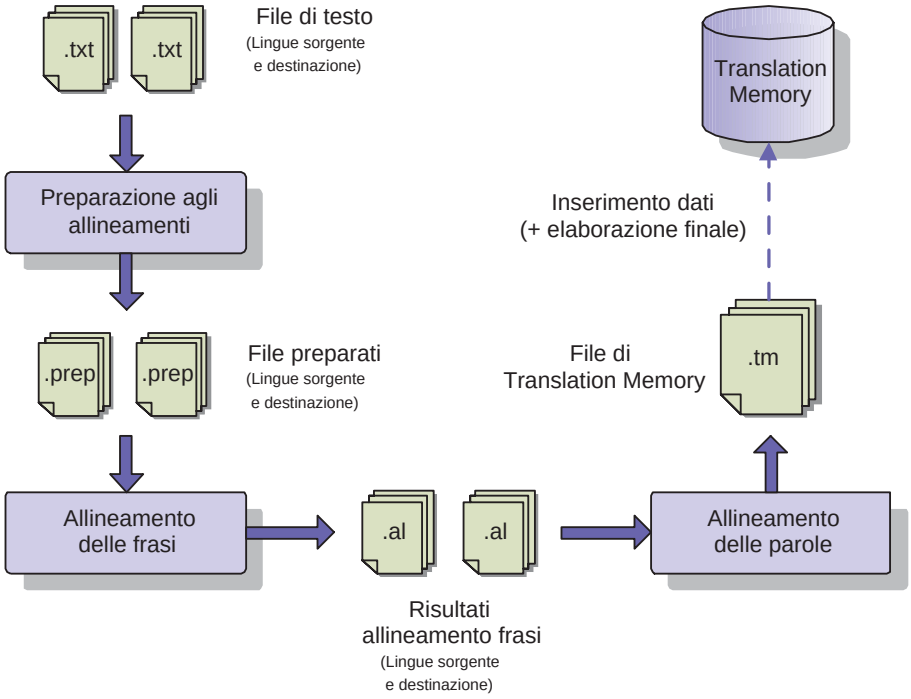


Figura 4.19: Creazione della Translation Memory: schema riassuntivo

In realtà, prima dell'inserimento vero e proprio viene svolta un'ultima elaborazione, analoga a quella già descritta nel caso di file da pretradurre (sezione 3.4.1). In particolare, si provvede a generare un'ulteriore versione della frase sorgente, quella che verrà esaminata dagli algoritmi di ricerca di similarità visti nel capitolo precedente: a seconda delle preferenze dell'utente, l'elaborazione consisterà nella semplice rimozione della punteggiatura o nello stemming.

Analogamente a quanto si è visto per l'allineamento delle parole tra lingua sorgente e lingua destinazione, anche l'operazione di stemming è stata estesa per supportare la ricerca di sottoparti: utilizzando una nuova stringa di "coordinate" è possibile comprendere a quali parole della frase originale corrispondano le parole presenti nella parte normalizzata (con stemming).

Per ottenere questi risultati, è stato esteso il package di stemming sviluppato da F. Gavioli [13], ottenendo come risultato non solo la frase normalizzata, ma anche la stringa contenente le posizioni (numeri) delle parole della frase originaria cui corrispondono le parole della frase normalizzata (per una discussione completa delle migliorie apportate si veda il prossimo capitolo).

Ad esempio, lo stemming di "This means you can collect a library of images" restituisce non solo la frase "mean can collect library image", ma anche la stringa di posizioni "< 2 >< 4 >< 5 >< 7 >< 9 >" (la prima parola della frase normalizzata corrisponde alla seconda parola della frase originaria, la seconda alla quarta, e così via), permettendo la futura corretta estrazione dei risultati.

A questo punto si dispone di tutti i campi presenti nella tabella della Translation Memory (descritti nella sezione 3.3) ed è possibile completare il processo di creazione della TM inserendo i dati nel DataBase.

4.6 Pretraduzione

In EXTRA l'operazione di pretraduzione consiste nell'analizzare un documento e produrre in output due file (figura 4.20):

- un file contenente le varie frasi, opportunamente identificate e suddivise;
- un file contenente, per ciascuna di queste frasi, tutti i suggerimenti di traduzione trovati nell'operazione di ricerca di somiglianza (sia frasi intere sia sottoparti) e il relativo spazio per permettere al traduttore di inserire la traduzione vera e propria.

I due file presentano un formato del tutto analogo all'output del processo di allineamento di frasi (vedi sezione 4.3.6): questo per permettere al traduttore, dopo aver inserito le traduzioni corrette, di procedere con l'allineamento delle parole tra le due lingue e con l'inserimento finale del nuovo lavoro nella Translation Memory.

Le varie operazioni legate alla ricerca di somiglianza sono state già descritte nel Capitolo 3, cui si rimanda per un'analisi dettagliata. In particolare erano state descritte le operazioni relative alla ricerca di frasi intere e di sottoparti, con le relative query che arrivavano a registrare nelle tabelle dei risultati i dati dei vari match trovati.

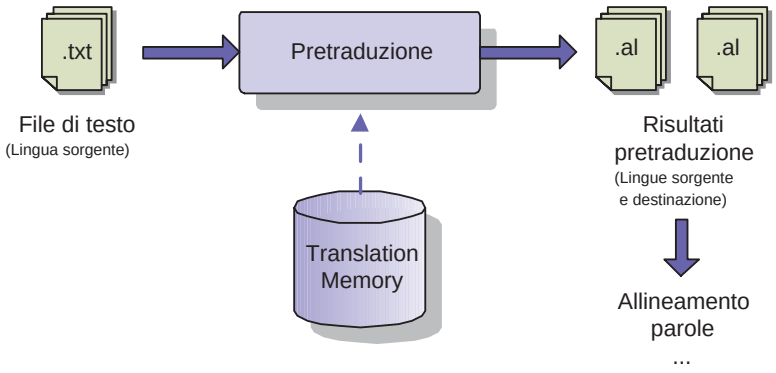


Figura 4.20: Pretraduzione: schema

Rimane pertanto solo da descrivere l'estrazione di questi dati dalle tabelle e la presentazione dei suggerimenti derivanti.

4.6.1 La query di estrazione dei risultati

Per estrarre i dati relativi ai match delle frasi intere e delle sottoparti si utilizza una query unica, formata da una parte per i *fullmatch* (match tra frasi intere), sempre presente, e da una opzionale per i *submatch* (match tra sottoparti). I dati estratti per i *fullmatch* sono i seguenti:

- *Codice2* e *Codice1*: i codici, rispettivamente, della frase da tradurre e di quella della Translation Memory riconosciuta come simile;
- *Frase2* e *Frase1*: le frasi intere elaborate (normalizzate o semplicemente senza punteggiatura);
- *FraseOrig2* e *FraseOrig1*: le frasi in forma originaria (non elaborata);
- *FraseTrad2* e *FraseTrad1*: le frasi tradotte nella lingua destinazione;
- *Dist*: la distanza assoluta tra la coppia di frasi simili (il valore dell'edit distance);
- *DistRel*: la distanza tra le frasi simili espressa come percentuale della lunghezza della frase cercata.

La parte di query relativa alle frasi intere è strutturata come segue.

```
                                    -- estrazione risultati frasi intere
SELECT fm.cod2 AS codice2,
        r2.frase AS frase2, '' AS sfrase2,
        r2.fraseorig AS fraseorig2, '' AS sfraseorig2,
    fm.cod1 AS codice1,
        r1.frase AS frase1, '' AS sfrase1,
        r1.fraseorig AS fraseorig1, '' AS sfraseorig1,
        r1.frasetrad AS frasetrad1, '' AS sfrasetrad1,
    fm.dist AS dist, (fm.dist / r2.wordLen) AS distrel,
        0 AS inizio, 0 AS sub
FROM FULLMATCH fm, <tab1> r1, <tab2> r2
WHERE fm.cod1 = r1.codice
AND fm.cod2 = r2.codice
ORDER BY codice2, sub, inizio, dist, codice1
```

In caso di ricerca di sottoparti attiva, per i match tra sottoparti vengono anche estratti i seguenti dati:

- *SFrase2* e *SFrase1*: le parti delle frasi elaborate effettivamente riconosciute come simili;
- *Inizio*: il punto di inizio (nella frase cercata) della parte trovata;
- *SFraseOrig2* e *SFraseOrig1*: le parti delle frasi originarie corrispondenti;
- *SFraseTrad2* e *SFraseTrad1*: le parti delle traduzioni corrispondenti.

Alla query viene aggiunta la seguente parte:

```
                                    -- eventuale estrazione risultati sottoparti
UNION SELECT sm.cod2 AS codice2,
        r2.frase AS frase2,
        wordSubString(r2.frase, sm.i2, sm.f2) AS sfrase2,
        r2.fraseorig AS fraseorig2,
        wordSubString(r2.fraseorig, transPos(r2.spos,sm.i2),
            transPos(r2.spos,sm.f2)) AS sfraseorig2,
    sm.cod1 AS codice1,
        r1.frase AS frase1,
        wordSubString(r1.frase, sm.i1, sm.f1) AS sfrase1,
```

```
        r1.fraseorig AS fraseorig1,
        wordSubString(r1.fraseorig, transPos(r1.spos,sm.i1),
            transPos(r1.spos,sm.f1)) AS sfraseorig1,
        r1.frasetrad AS frasetrad1,
        wordSubString(r1.frasetrad,transPos(r1.apos,transPos(r1.spos,sm.i1)),
            transPos(r1.apos,transPos(r1.spos,sm.f1))) AS sfrasetrad1,
    sm.dist AS dist, (sm.dist / (sm.f2−sm.i2+1)) AS distrel,
        sm.i2 AS inizio, 1 AS sub
FROM SUBMATCH sm, <tab1> r1, <tab2> r2
WHERE sm.cod1 = r1.codice
AND sm.cod2 = r2.codice
```

Come si vede le due parti sono perfettamente compatibili: quella relativa alle frasi intere estrae campi vuoti per i dati che esistono solo per le sottoparti; mediante il valore del campo aggiuntivo *sub* si è in grado di stabilire se i match trovati riguardano frasi intere ($sub = 0$) o sottoparti ($sub = 1$).

La funzione *wordSubString*, come già visto nelle query di ricerca, permette di estrarre una sottoparte di una frase, data la posizione della parola iniziale e di quella finale. La funzione *transPos* è invece nuova: essa esegue la conversione tra le coordinate delle varie frasi, permettendo di risalire dalla sottoparte di frase normalizzata alla sottoparte di frase originaria, e dalla sottoparte di frase originaria alla sottoparte di frase tradotta, utilizzando i dati di stemming ed allineamento presenti nella Translation Memory (figura 4.21).

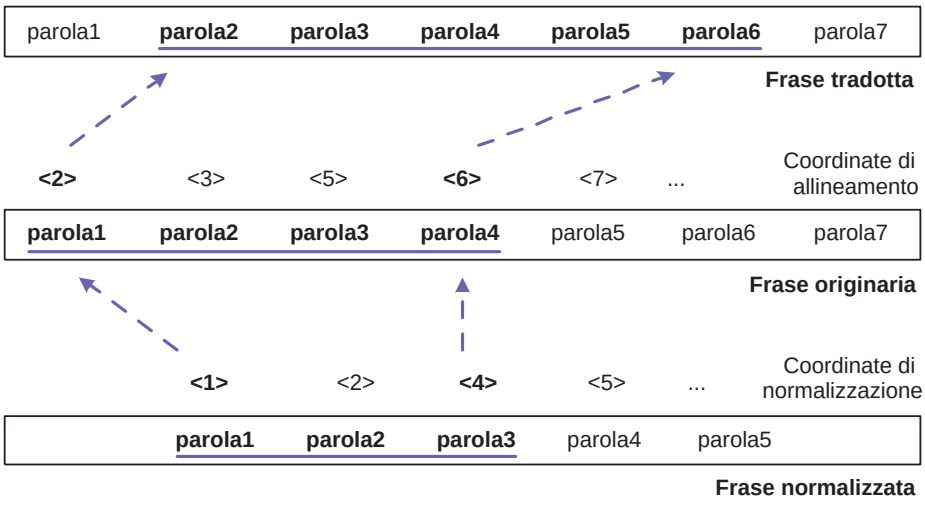


Figura 4.21: Estrazione risultati: le coordinate per le sottoparti

Per quanto riguarda l'ordinamento, i match vengono ordinati in modo da presentare le frasi cercate nell'ordine originario; per ciascuna di esse vengono proposti i match completi (partendo dal più simile) o i match tra sottoparti (ordinati per punto di inizio nella frase cercata, quindi per similarità).

4.6.2 La presentazione dei risultati

A pretraduzione avvenuta, vengono generati i file di output visti; un esempio di file di pretraduzione con i suggerimenti trovati è il seguente:

```
## PHRASE FOUND:     (COD 153) check fault occur get touch nearest
##                             sale service
##                   (ORIG)    If after the above checks the fault
##                             still occurs, get in touch with the
##                             nearest After Sales Service.
## FULL(D=0/RD=0,00):(COD 11468) check fault occur get touch nearest
##                               sale service
##                   (ORIG)      If after the above checks the fault
##                               still occurs, get in touch with the
##                               nearest After Sales Service
##                   (TRAD --> ) Se dopo i suddetti controlli il
##                               piano o il forno non funzionano
##                               correttamente, contattare il Servizio
##                               Assistenza piu' vicino
##
<Insert translation here>
.DIST 0
.SENT
```

Da notare che il formato utilizzato consente di effettuare, dopo l'intervento del traduttore, il successivo allineamento delle parole: in particolare, i suggerimenti non interferiscono con le successive elaborazioni poiché inseriti come linee di commento (`##`).

Oltre ai vari suggerimenti, nei file vengono anche inserite utili informazioni aggiuntive, quali tempo utilizzato per le varie fasi della ricerca, numero di frasi trovate, e così via. Queste informazioni vengono per comodità riepilogate anche

in una finestra grafica, che viene mostrata al termine del processo e permette all'utente di cogliere con un colpo d'occhio le prestazioni e l'efficacia della pretraduzione appena svolta (figura 4.22).

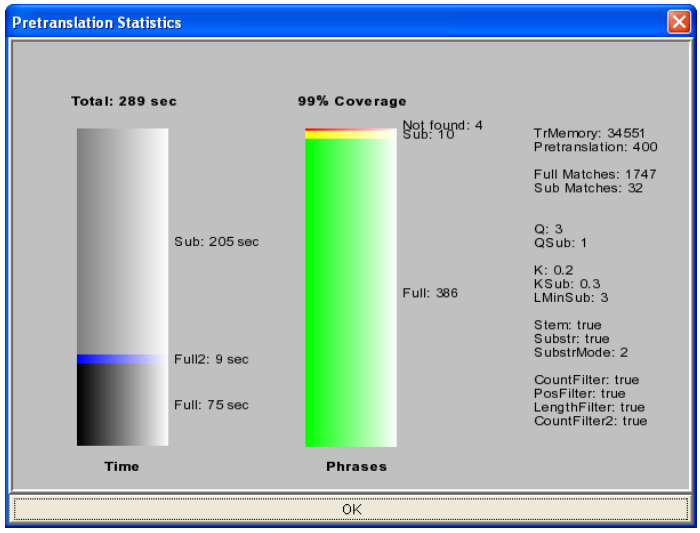


Figura 4.22: EXTRA: il resoconto grafico sulla pretraduzione

In particolare viene mostrato: il tempo impiegato, sia parziale sia totale, la *copertura* dei suggerimenti della pretraduzione (numero di frasi trovate per intero, numero di frasi per cui si è trovata almeno una sottoparte, numero di frasi non trovate, numero di suggerimenti trovati), numero di frasi presenti nella Translation Memory, numero di frasi nel file pretradotto, e il valore dei principali parametri che caratterizzano la ricerca.

4.7 Analisi della Translation Memory

A completamento delle principali funzioni, in EXTRA sono proposti anche un paio di strumenti che permettono di verificare lo stato della Translation Memory e di compierne una semplice analisi statistica. È possibile:

- verificare il numero di frasi nella Translation Memory;
- visualizzare automaticamente il grafico di distribuzione delle lunghezze delle varie frasi presenti, permettendo così all'utente di capire il carattere (esponenziale, uniforme, ...) dei dati che sta utilizzando, ad esempio, per testare il sistema.

In figura 4.23 è mostrata la distribuzione ottenuta da uno dei dataset utilizzati nelle prove.

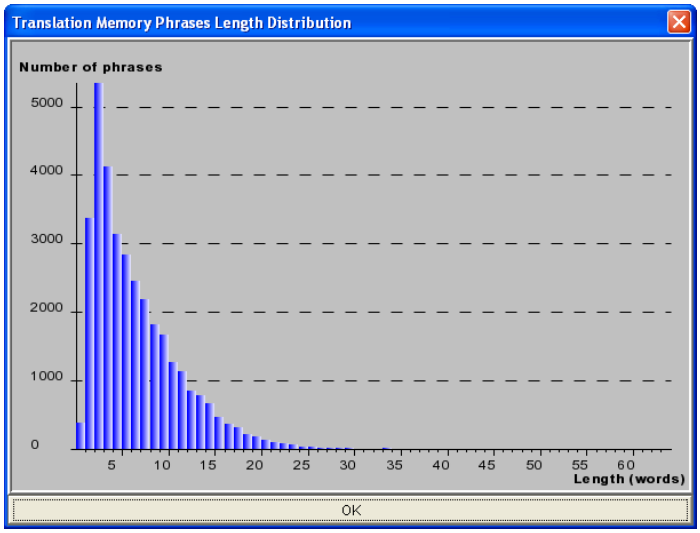


Figura 4.23: EXTRA: l'analisi grafica della distribuzione della TM

4.8 Configurazione

EXTRA dispone di un comando di configurazione, che fornisce una serie di finestre in cui possono essere impostati tutti i parametri che si sono descritti per le varie funzioni: quelli per la ricerca di similarità (descritti nella sezione 3.10),

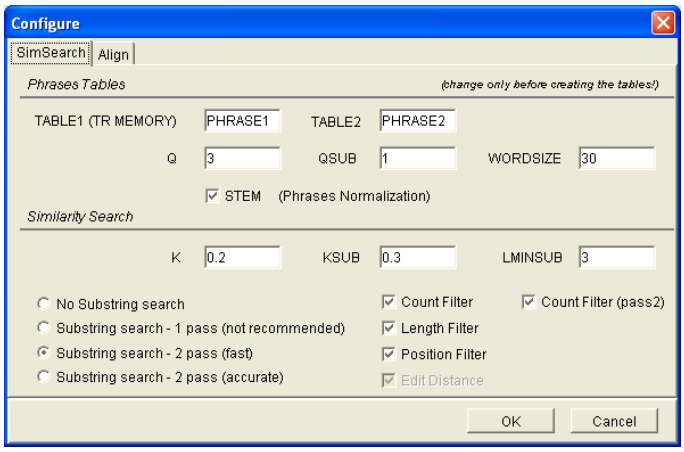


Figura 4.24: EXTRA: la configurazione della ricerca di similarità

e quelli per le funzioni di allineamento (delle frasi, sezione 4.3.5, e delle parole, sezione 4.4.6).

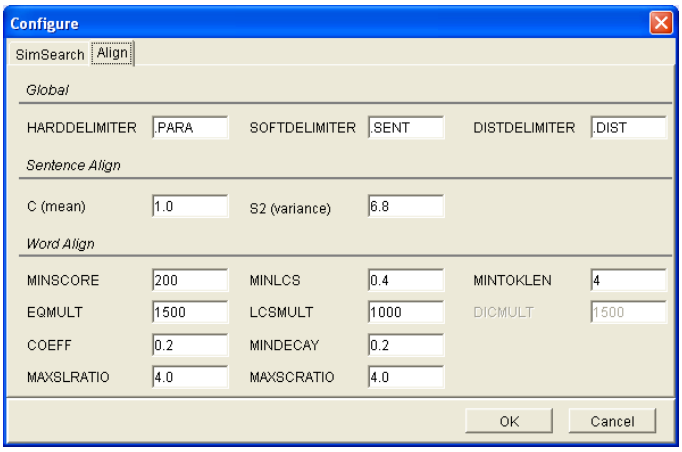


Figura 4.25: EXTRA: la configurazione degli allineamenti

Capitolo 5

Il progetto del software

Nei due capitoli precedenti si sono presentate le funzionalità di EXTRA e gli approcci seguiti per realizzarle, spiegandone le idee alla base e mantenendole quanto più possibile svincolate dalla loro effettiva implementazione.

In questo capitolo verrà allora approfondito proprio l'aspetto del progetto e dell'implementazione: verranno mostrati, mediante package e class diagram UML, i package e le classi che compongono il programma, e ci si spingerà più a basso livello, con i Data Flow Diagram, per analizzare in dettaglio come i metodi delle varie classi si rapportino tra loro per realizzare le principali funzionalità presenti.

5.1 I package e le classi

EXTRA è organizzato 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 quello 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.

Nel package diagram di figura 5.1 sono mostrate le dipendenze: il package *Main* "chiama" i package *SimSearch* e *Align*, che forniscono le principali funzionalità del software. A loro volta, questi due package si avvalgono del package *GlobalUtility*; infine, *Stemming* è utilizzato da *SimSearch* per normalizzare le frasi su cui poi effettuare la ricerca di somiglianza.

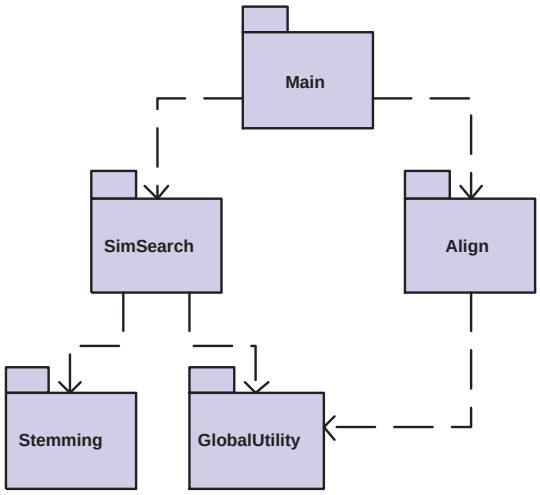


Figura 5.1: EXTRA: package diagram

Nelle sezioni che seguono verranno mostrate le classi che formano ciascuno di questi package: mediante class diagram verranno evidenziati, oltre ai metodi e alle variabili membro, anche i rapporti di dipendenza tra le varie classi. Per un approfondimento ulteriore, si ricorda che nella sezione 5.2 sono mostrati i relativi DFD, mentre nell'appendice A sono riportate le parti fondamentali del codice Java.

5.1.1 Package SimSearch

La classe principale di questo package è l'omonima *SimSearch*, che contiene i metodi che realizzano le funzionalità principali offerte da questo package: la pretraduzione (metodo *pretranslate()*), in cui viene effettuata la ricerca di similarità, la gestione delle tabelle (*createTables()*, *dropTables()*), l'analisi della Translation Memory (*memDistr()*, *memStatus()*), l'aggiunta di dati alla Translation Memory (metodo *addMemory()*).

Questa classe si avvale delle altre mostrate in figura 5.2: *PretranslationStat* e *Result*, istanziabili e utilizzate per implementare, rispettivamente, le statisti-

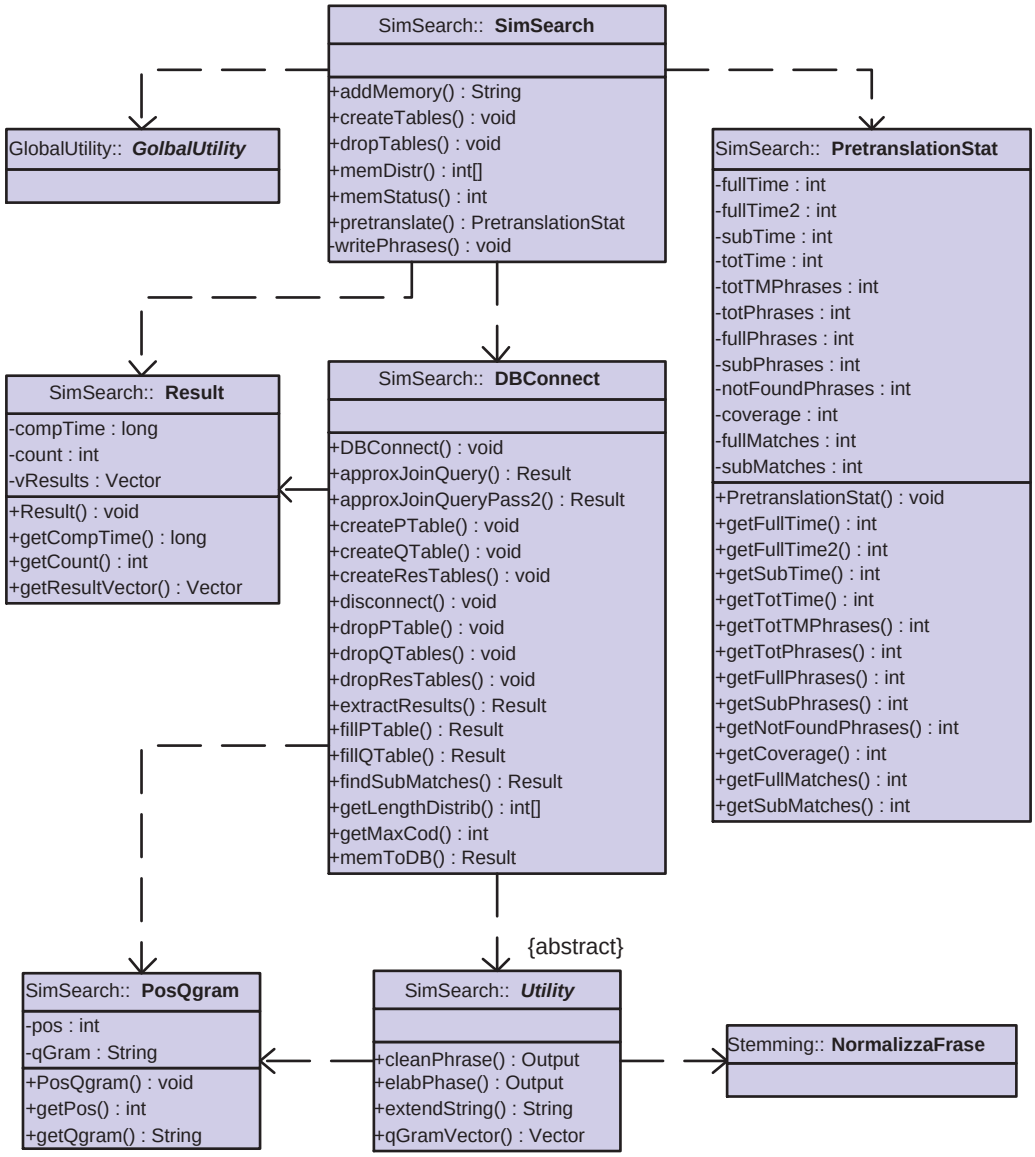


Figura 5.2: Package SimSearch: class diagram (parte 1)

che di pretraduzione, mostrate alla fine dell'operazione, e il risultato di una elaborazione (completo di tempo impiegato). Essa utilizza anche e soprattutto *DBConnect*, che contiene tutti i metodi che si interfacciano con il DBMS Oracle, tra cui le fondamentali query di ricerca di similarità, descritte nel capitolo 3 e mostrate per esteso nell'appendice B.3. Queste query vengono realizzate da metodi quali *approxJoinQuery()* (ricerca frasi intere) o *findSubMatches()* (ricerca sottoparti), che utilizzano istruzioni JDBC [29] per interfacciarsi ad Oracle ed eseguire query non definite staticamente, ma create dinamicamente.

La classe *PosQGram* implementa l'oggetto q-gramma posizionale, utilizzato nelle fasi di filtraggio della ricerca; la classe astratta *Utility* contiene una serie di funzioni di varia utilità, come la generazione di un vettore di q-grammi da una frase (metodo *qGramVector()*) o l'elaborazione delle frasi stesse. Questa classe si avvale anche, per la normalizzazione, del package *Stemming*, le cui classi saranno analizzate nella relativa sezione.

Completano il package le classi astratte *Distance* e *DBUtility* (figura 5.3), che contengono i metodi legati al calcolo dell'edit distance e altre funzioni di utilità. Queste funzioni si differenziano da quelle presenti nella classe *Utility* per il fatto che saranno richiamate direttamente dalle query eseguite in Oracle; a questo fine, devono essere registrate e caricate nel DBMS mediante l'apposito comando *loadjava* e i relativi script (mostrati nell'appendice B.1). Esse divengono in pratica funzioni interne al DBMS e vengono pertanto definite "Stored Procedure" [32]. Tra quelle presenti ricordiamo *wordSubString()*, per estrarre una sottoparte di una frase, *wordLen()*, per calcolare la lunghezza di una frase (in parole), *transPos()*, per il calcolo delle corrispondenze tra sottoparti di frasi nelle due lingue.

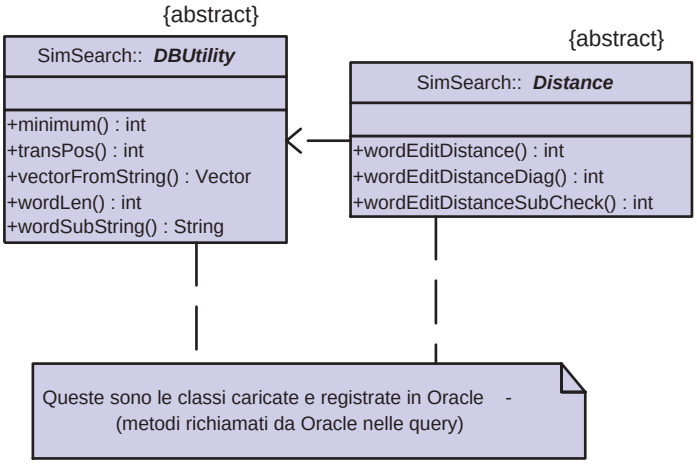


Figura 5.3: Package SimSearch: class diagram (parte 2)

Si ricorda che nei DFD delle sezioni 5.2.1 e 5.2.5 verrà mostrato più in dettaglio il funzionamento dei metodi per la ricerca di similarità e per l'aggiunta di dati alla Translation Memory. Infine, una nota per quanto riguarda le tabelle utilizzate da questo package: per una descrizione completa si rimanda alla sezione 3.3.

5.1.2 Package Align

In figura 5.4 sono mostrate le classi del package *Align* che realizzano l'allineamento tra frasi: si tratta di una classe astratta, *SentAlign*, contenente i metodi che implementano gli algoritmi di allineamento descritti nel capitolo 4, e *SentAlignment*, una classe istanziabile che rappresenta l'oggetto di allineamento vero e proprio.

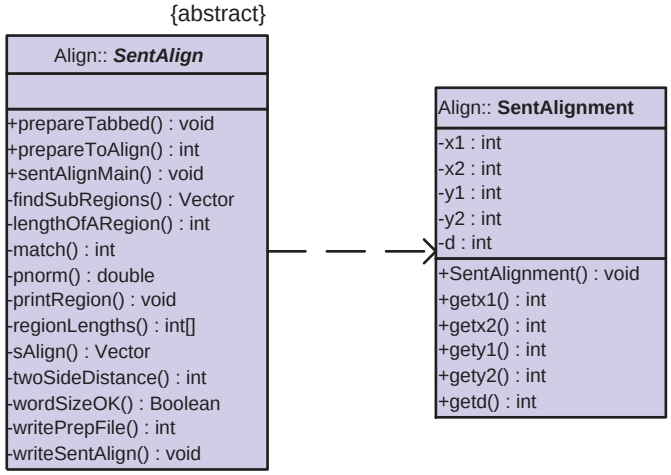


Figura 5.4: Package Align: class diagram (parte 1)

Discorso analogo per quanto riguarda l'allineamento delle parole (figura 5.5):

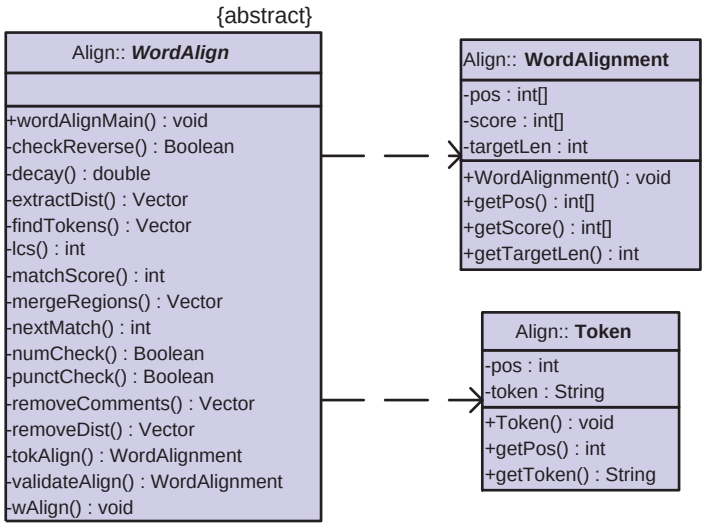


Figura 5.5: Package Align: class diagram (parte 2)

La classe astratta *WordAlign* implementa gli algoritmi, *WordAlignment* è l'oggetto di allineamento tra le parole di una coppia di frasi, *Token* è l'oggetto da allineare, sia esso una parola od un elemento di punteggiatura.

Per maggiori dettagli sull'utilizzo e la relazione tra i metodi di queste classi, si rimanda ai DFD delle sezioni 5.2.2 (preparazione all'allineamento), 5.2.3 (allineamento tra frasi), e 5.2.4 (allineamento tra parole).

5.1.3 Package Stemming

Il package di ricerca di somiglianza si avvale dei servizi del package *Stemming*, sviluppato originariamente da F. Gavioli [13], per effettuare la normalizzazione delle frasi. In questa sezione verranno mostrate le classi e le tabelle di questo package, in modo da creare un quadro unitario del software nel suo insieme; inoltre verranno introdotte le estensioni e le ottimizzazioni che sono state apportate per soddisfare le nuove esigenze di utilizzo, sia in termini di nuove funzioni sia in termini di maggiore efficienza.

Estensioni al codice

Le classi del package *Stemming* utilizzate in questo software sono mostrate in figura 5.9. La classe *NormalizzaFrase* (e in particolare il metodo *normalizz()*) è il cuore del package ed è quella che viene richiamata direttamente da *SimSearch*. Utilizzando le classi *Rule* (per le regole di elaborazione) e le classi radice *Rad* (per nomi, aggettivi e avverbi), essa restituisce un oggetto *Output*, che prevede: *phrase*, una stringa contenente la frase normalizzata, e *comp_time*, il tempo impiegato per l'elaborazione. Proprio la classe dei risultati, *Output*, è stata estesa per coprire tutte le esigenze della nuova applicazione: il software, nella ricerca di sottoparti simili, non poteva comprendere a quali parole della frase originale corrispondessero le parole presenti nella parte normalizzata. E' stato pertanto aggiunto un membro a questa classe: *sPos*, una stringa contenente le posizioni (numeri) delle parole della frase originaria cui corrispondono le parole della frase normalizzata (vedi anche la sezione 4.5).

Ottimizzazioni alle tabelle e alle query

Le tabelle utilizzate dallo stemmer sono molto semplici e vengono mostrate in figura 5.7. Esse vengono utilizzate fondamentalmente per controllare se una

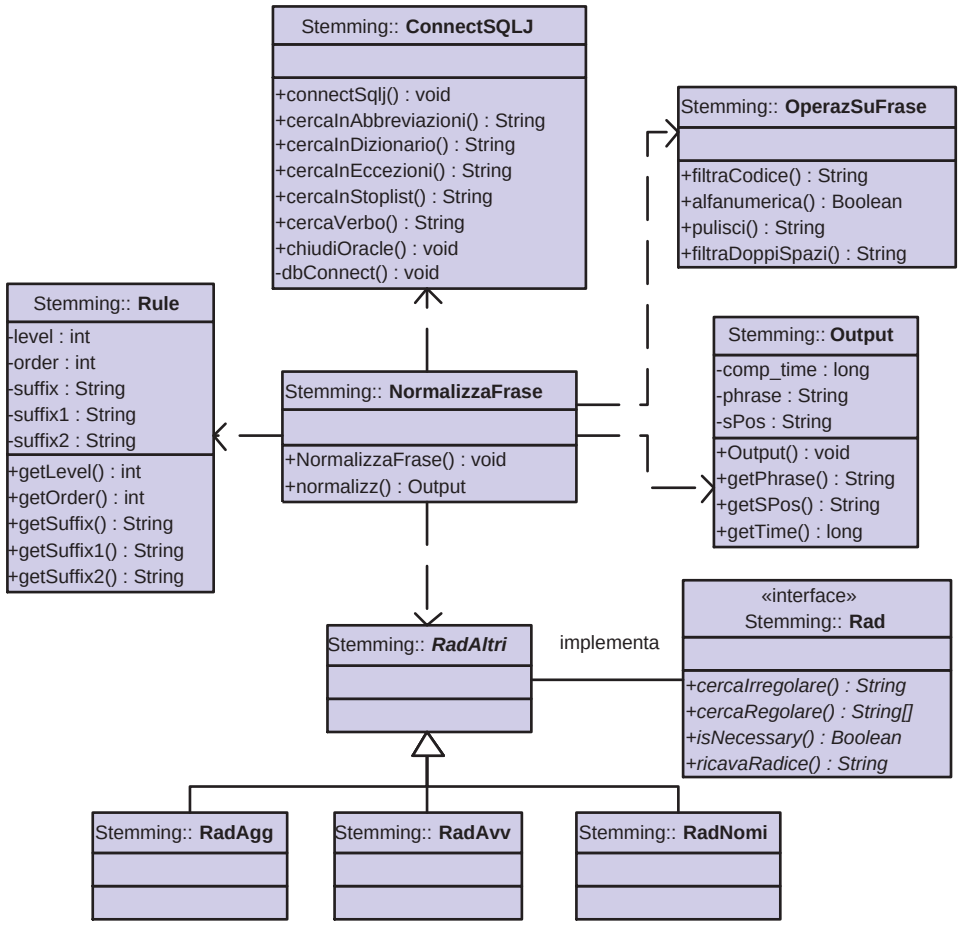


Figura 5.6: Package Stemming: class diagram

parola appartiene ad una determinata categoria (stopword nella tabella *STOPLIST*, verbi nella tabella *VERBI_EN*), se una parola normalizzata compare nel dizionario inglese (tabella *DIZ*), o se è un'abbreviazione (tabella *ABBREV_EN*).

Le tabelle non presentavano chiavi primarie (e quindi i dati non erano ben organizzati) e non disponevano di indici adatti a velocizzare la ricerca. Pertanto sono stati aggiunti:

- tabelle *DIZ* e *STOPLIST*: chiave e indice sulla colonna *word*, in modo da velocizzarne la ricerca ed impedire al tempo stesso la presenza di due parole uguali (caso che capitava erroneamente);

- tabella *VERBI_EN*: chiave sugli id (in modo che dato id del verbo e id

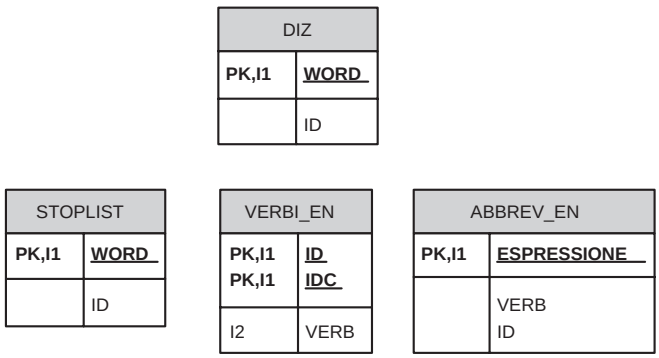


Figura 5.7: Package Stemming: le tabelle utilizzate

della coniugazione corrispondesse uno e un solo termine) e indici su di essi ma anche sulla colonna *verb*, contenente i termini ricercati;

- tabella *ABBREV_EN*: chiave e indice sulla colonna *espressione* (quella ricercata nelle query).

Inoltre è stato leggermente modificato ed ottimizzato il codice delle query ed è stato corretto un malfunzionamento nella query dei verbi, che si occupa di controllare se una parola è un verbo e di coniugarla all'infinito: essa generava errore nel caso di ricerca di un termine collegato a differenti id (ad esempio "bit", sia infinito del verbo omonimo sia passato di "bite").

Con queste migliorie l'operazione di stemming ha beneficiato di una maggiore correttezza e di un notevole incremento (70-80%) di prestazioni.

5.1.4 Package GlobalUtility

Nel package *GlobalUtility* sono racchiuse quelle funzioni di utilità comuni a tutto il software, utilizzate per la ricerca di similarità ma anche per gli allineamenti (figura 5.8).

Queste funzioni sono contenute nella classe omonima *GlobalUtility*: le utilità per la lettura dei file, sia riga per riga (*readLines()*) sia identificando correttamente le varie frasi (*readPhrases()*), quelle per la separazione della punteggiatura (*separatePunct()*), ed altre ancora. Inoltre abbiamo la funzione per la lettura delle informazioni di una Translation Memory (*readTMFile()*), insieme alla relativa classe $TMPhrase$ che unisce le informazioni stesse in un unico oggetto.

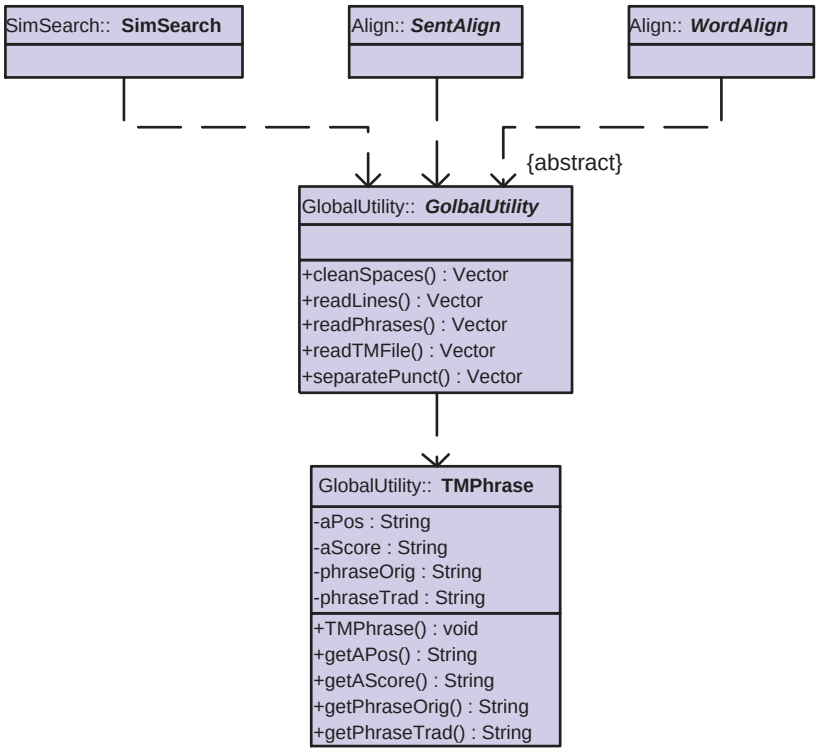


Figura 5.8: Package GlobalUtility: class diagram

5.1.5 Package Main

Le classi del package Main costituiscono lo scheletro del programma e ne realizzano l'interfaccia: esse non implementano le funzionalità vere e proprie, ma piuttosto utilizzano ed uniscono quelle fornite dai package principali, già analizzati. Per questo motivo le classi di questo pacchetto vengono mostrate con un approfondimento molto minore, senza scendere nel dettaglio dei vari metodi presenti: essi sono piuttosto numerosi e, come spiegato, non troppo significativi ai fini di questa analisi.

In figura 5.9 sono mostrate le principali classi presenti, e le relative dipendenze: *Extra*, la classe principale, *MainFrame*, la classe che realizza la finestra grafica principale dell'applicazione, *MainFrame_CongifureDialog* e *MainFrame_GraphicDialog*, le classi che implementano le finestre figlie, rispettivamente quella di configurazione (descritta in 4.8) e quella per i grafici. Quest'ultima è utilizzata per visualizzare sia le statistiche di pretraduzione (sezione 4.6.2) sia la distribuzione delle frasi presenti nella Translation Memory (sezione 4.7).

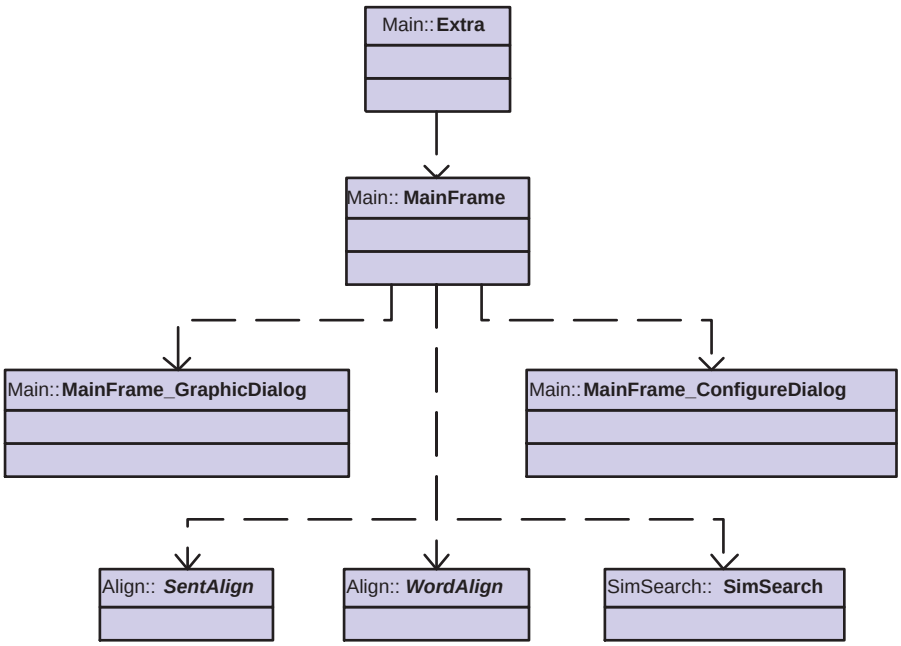


Figura 5.9: Package Main: class diagram

5.2 I flussi di dati (DFD)

Per meglio spiegare e riassumere lo schema di funzionamento del programma si riportano i *Data Flow Diagram (DFD)* riguardanti le operazioni più significative: la ricerca di similarità e gli allineamenti. Per migliorarne la comprensione e non appesantirli eccessivamente, in alcuni casi essi risultano leggermente semplificati rispetto alla effettiva implementazione, pur rimanendo fedeli alla logica sottostante. Per quanto riguarda i metodi interessati, vengono indicati, oltre al nome degli stessi, anche la classe di appartenenza (anteposta) e il relativo package (in basso tra parentesi).

5.2.1 Ricerca di similarità

Ricerca di frasi intere

I diagrammi di questa sezione mostrano i metodi utilizzati per compiere le operazioni di ricerca tra frasi intere, come descritte nella sezione 3.6.

In figura 5.10 vengono mostrati le tabelle del DB interessate, e, ad alto livello, il metodo che gestisce la query, *approxJoinQuery()*: come si vede esso è richia-

mato dalla funzione di pretraduzione, di cui la ricerca di similarità costituisce una parte fondamentale.

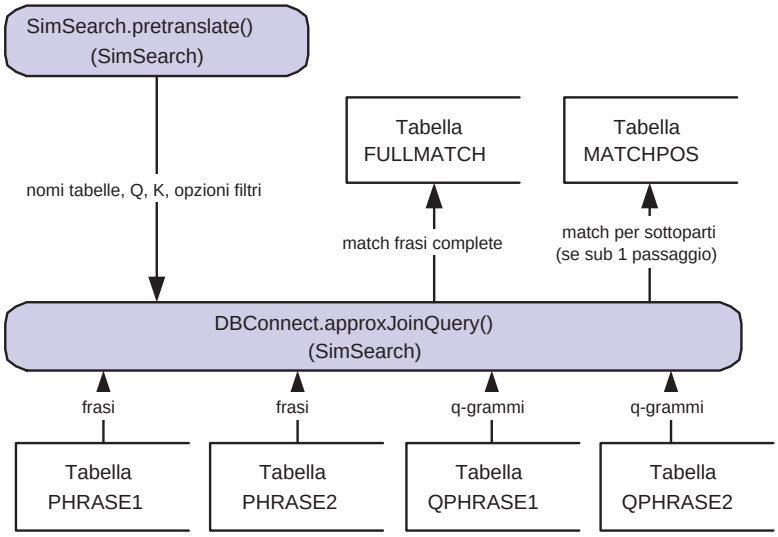


Figura 5.10: Ricerca frasi intere simili: data flow diagram

In figura 5.11 si presenta invece più in dettaglio il punto di vista del DBMS, con le *stored procedure* da esso utilizzate per risolvere la query.

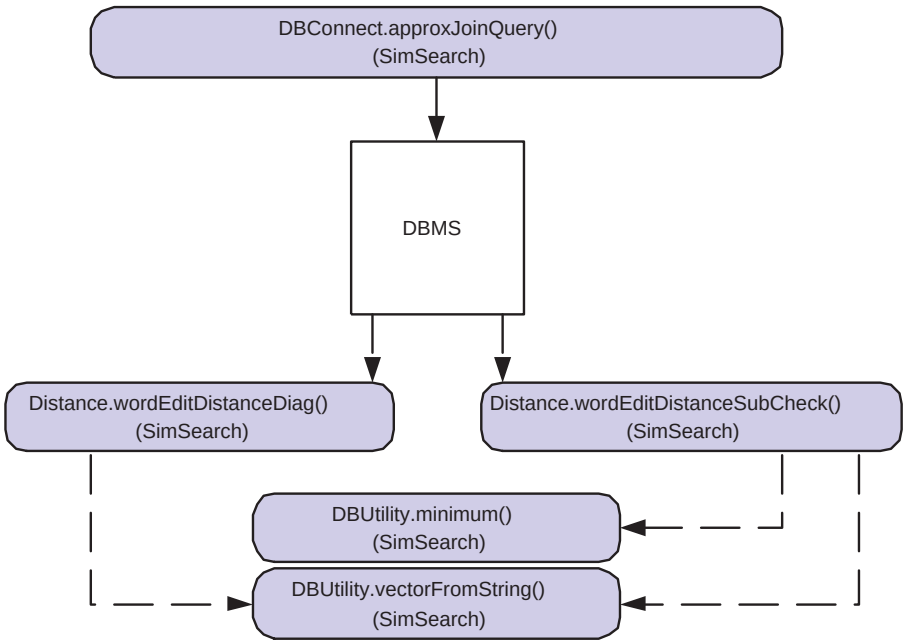


Figura 5.11: Ricerca frasi intere simili (DMBS): data flow diagram

Per queste ultime, vengono rappresentate anche le semplici funzioni di utilità da esse richiamate:

- *minimum()*, utilizzata dall'algoritmo standard dell'edit distance per trovare il minore tra tre valori numerici;
- *vectorFromString()*, utilizzata da entrambi gli algoritmi di edit distance per separare e confrontare le parole di una frase.

Gli schemi riguardanti la query di preparazione alle sottoparti sono sostanzialmente analoghi e non vengono pertanto presentati.

Ricerca di sottoparti

La ricerca di sottoparti, come descritta nella sezione 3.9, utilizza i metodi e le tabelle mostrati in figura 5.12; la funzione che esegue la query vera e propria è *findSubMatches()*.

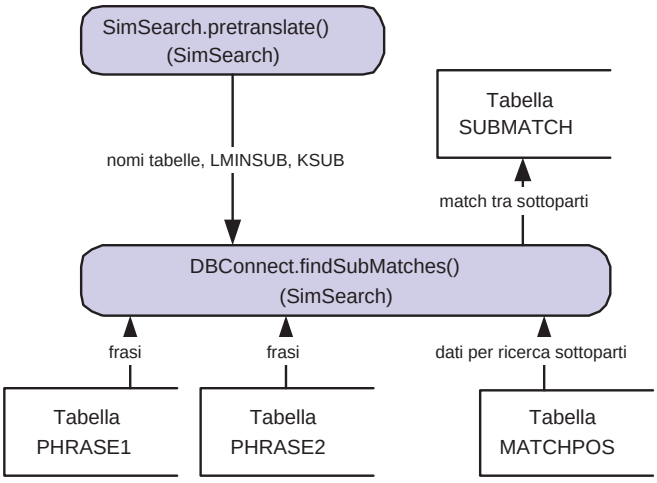


Figura 5.12: Ricerca sottoparti simili: data flow diagram

Per quanto riguarda le *stored procedure* richiamate dal DBMS, esse sono mostrate in figura 5.13: come per le frasi intere viene utilizzata la funzione per il calcolo dell'edit distance ottimizzato per diagonali. Inoltre compare la funzione *wordSubString()*, che come già ricordato estrae dalla frase la sottoparte interessata.

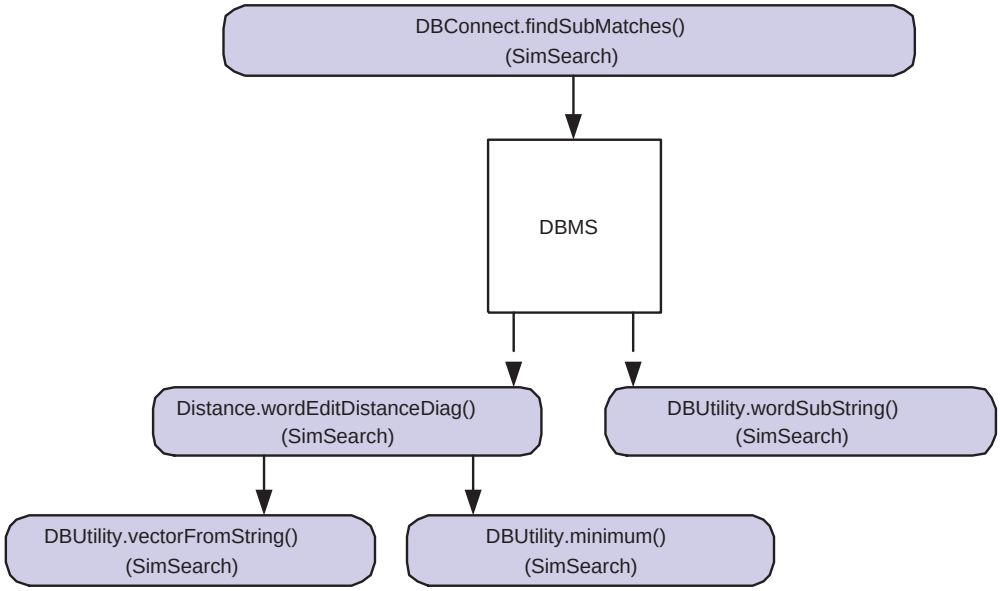


Figura 5.13: Ricerca sottoparti simili (DBMS): data flow diagram

Estrazione risultati

L'estrazione dei risultati utilizza ancora una volta una query, come descritto in 4.6.1. Questa query è implementata, mediante istruzioni JDBC [29], nel metodo *extractResults()*, che si occupa anche di produrre il file di output di tutto il processo di pretraduzione: questo contiene tutti i suggerimenti trovati e le informazioni aggiuntive quali tempo impiegato nelle varie fasi, percentuale di frasi trovate, ecc.

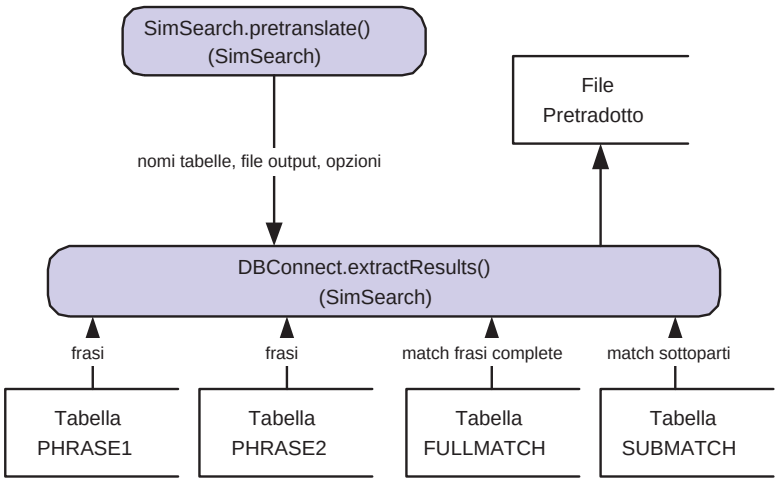


Figura 5.14: Estrazione risultati: data flow diagram

Per quanto riguarda i metodi richiamati dal DBMS: oltre a *wordSubString()* è utilizzato anche *transPos()*, per ricavare dalle stringhe di coordinate le giuste corrispondenze tra le frasi in lignua sorgente elaborate, quelle in forma originaria e quelle nella lingua destinazione.

5.2.2 Preparazione all'allineamento

La preparazione all'allineamento è descritta nella sezione 4.2. Nel DFD di figura 5.15 sono mostrati i metodi coinvolti: il metodo principale è *prepareToAlign()* della classe *Align*. Esso fa uso dei metodi della classe *GlobalUtility* per effettuare la lettura dei file da preparare:

- *readPhrases()*, nel caso in cui le frasi non siano già separate una per linea ed occorra pertanto identificarle correttamente;
- *readLines()* in caso contrario.

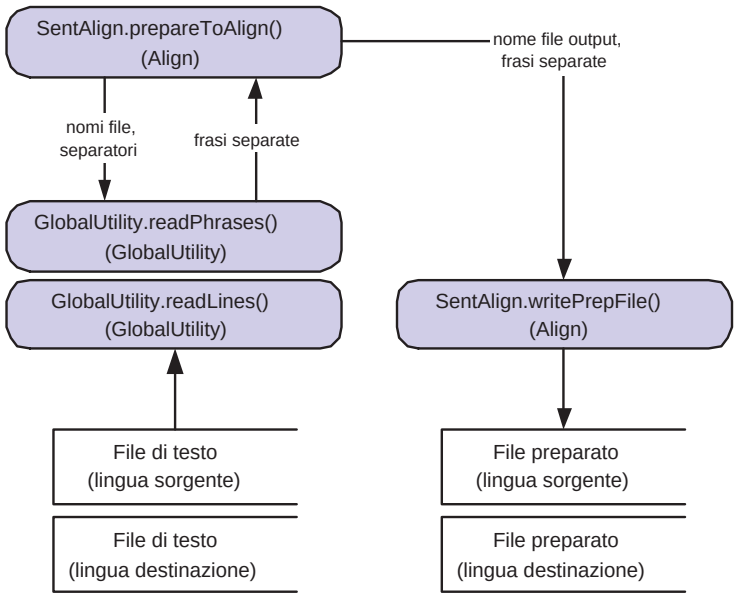


Figura 5.15: Preparazione all'allineamento: data flow diagram

Le frasi separate vengono dunque inviate al metodo *writePrepFile()*, che provvede a scriverle, nel formato opportuno, nei file preparati.

5.2.3 Allineamento delle frasi

I DFD di figura 5.16 e 5.17 mostrano i flussi di dati rigurdanti il processo di allineamento delle frasi, descritto nella sezione 4.3.

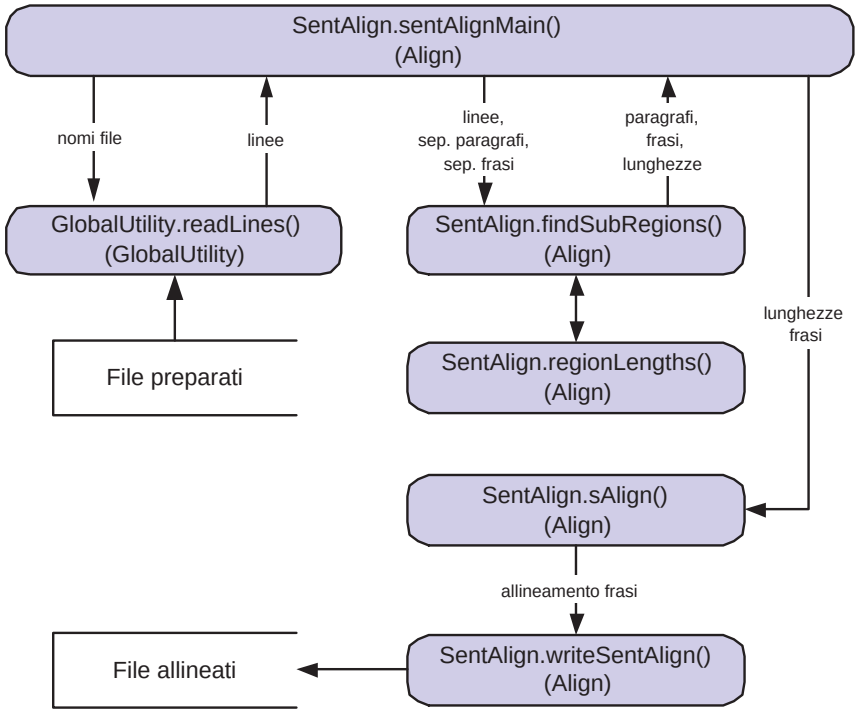


Figura 5.16: Allineamento frasi: data flow diagram (parte 1)

La funzione principale è *sentAlignMain()*: essa richiede la lettura del file di origine a *readLines()* del package *GlobalUtility*, quindi estrae dalle linee lette le regioni di testo (paragrafi, frasi) e le relative lunghezze tramite i metodi *findSubRegions()* e *regionLengths()*. Per ogni paragrafo viene costruito l'array contenente le lunghezze delle relative frasi e viene passato al metodo *sAlign()*, che esegue l'allineamento vero e proprio. Questo sarà poi scritto nel file tramite *writeSentAlign()*.

In figura 5.17 vengono dettagliate le funzioni utilizzate dal metodo di allineamento: *twoSideDistance()*, che implementa la funzione di distanza per i vari casi di allineamento, *match()*, che restituisce il punteggio tra due elementi da allineare, e *pnorm()*, utilizzata da *match()* per i calcoli riguardanti le distribuzioni normali. Insieme, queste funzioni forniscono i punteggi (distanze) su cui tutto il processo è basato.

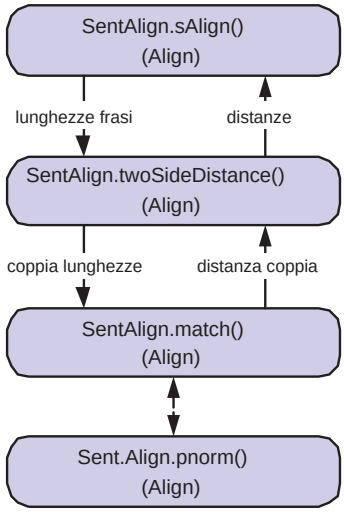


Figura 5.17: Allineamento frasi: data flow diagram (parte 2)

5.2.4 Allineamento delle parole

L'allineamento delle parole è descritto nella sezione 4.4; le figure 5.18 e 5.19 mostrano i relativi DFD.

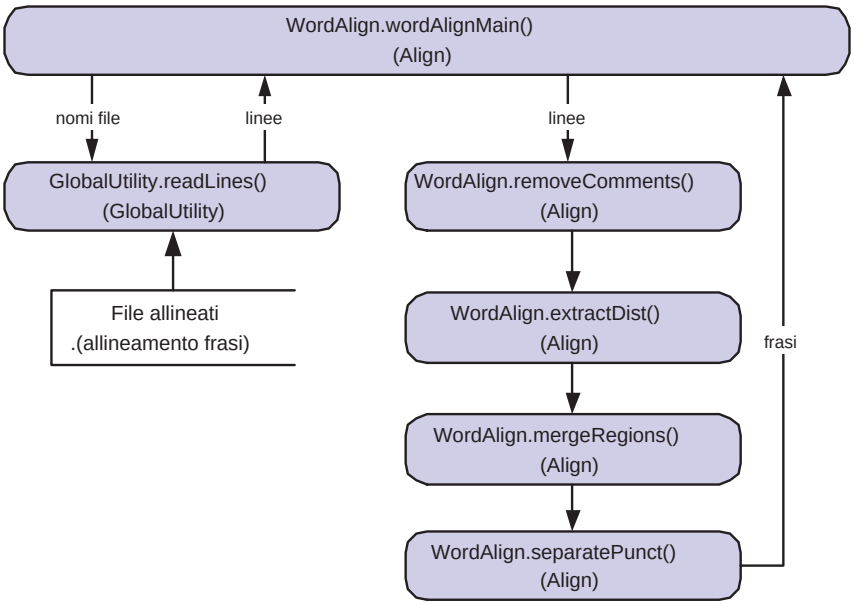


Figura 5.18: Allineamento parole: data flow diagram (parte 1)

La funzione principale è in questo caso *wordAlignMain()*: essa richiede alle consuete funzioni la lettura dei file di origine, in seguito esegue in catena una

serie di metodi per la successiva elaborazione dei dati estratti. In particolare, con *removeComments()* vengono rimosse le linee di commento, con *extractDist()* vengono estratte le informazioni di distanza prodotte dal precedente allineamento delle frasi, con *mergeRegions()* vengono fusi gli eventuali elementi formati da due frasi, ed infine con *separatePunct()* si provvede a separare la punteggiatura per la successiva corretta individuazione degli elementi da allineare (token).

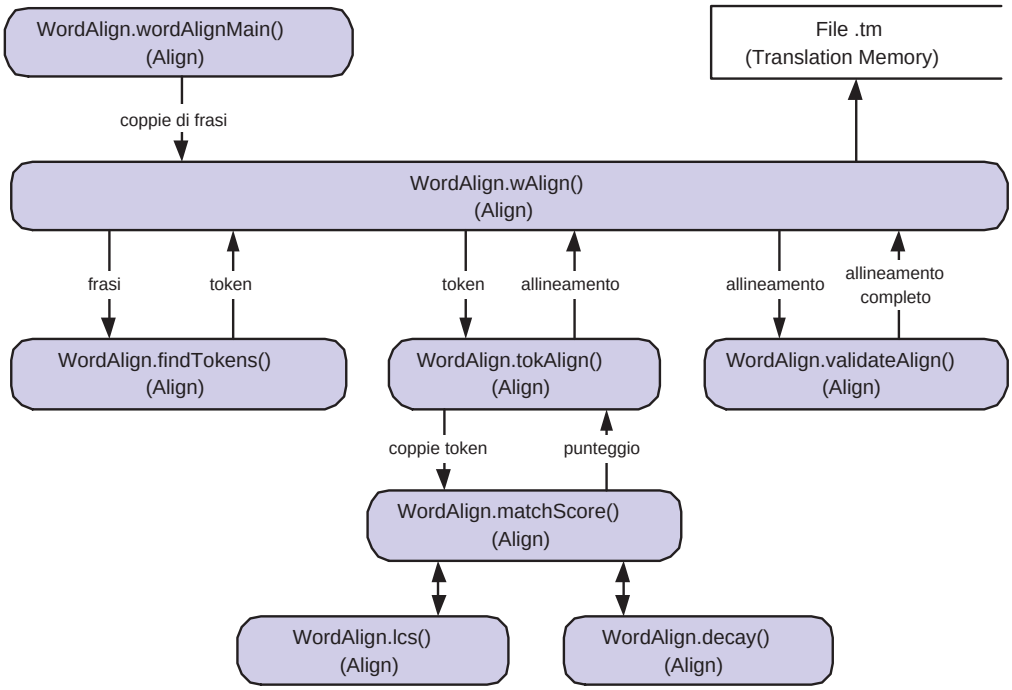


Figura 5.19: Allineamento parole: data flow diagram (parte 2)

Le frasi estratte vengono dunque passate alla funzione *wAlign()* (figura 5.19), che esegue l'algoritmo di allineamento vero e proprio utilizzando i metodi per il riconoscimento e la categorizzazione dei token (*findTokens()*) e per il loro allineamento (*tokAlign()*). L'assegnazione di un punteggio di allineamento ad ogni coppia di token è effettuata mediante le funzioni *macthScore()* e quelle da essa dipendenti. Infine, mediante *validateAlign()*, l'allineamento viene verificato e completato, per poi essere finalmente scritto nel file di output.

5.2.5 Aggiunta di dati alla TM

Il procedimento di aggiunta di dati alla Translation Memory è descritto nella sezione 4.5. Il metodo principale è in questo caso *addMemory()*, come mostrato

dalla figura 5.20: esso, tramite le consuete funzioni di utilità, legge i dati dal file *.tm*, generato dagli allineamenti, e mediante la funzione *memToDB()* provvede ad inserire i dati contenuti nel file nelle opportune tabelle, insieme alle elaborazioni delle varie frasi ottenute tramite *elabPhrase()*. A seconda delle impostazioni, queste elaborazioni consistono semplicemente nella rimozione della punteggiatura (metodo *cleanPhrase()*) o nella normalizzazione, ottenuta tramite il package *Stemming*.

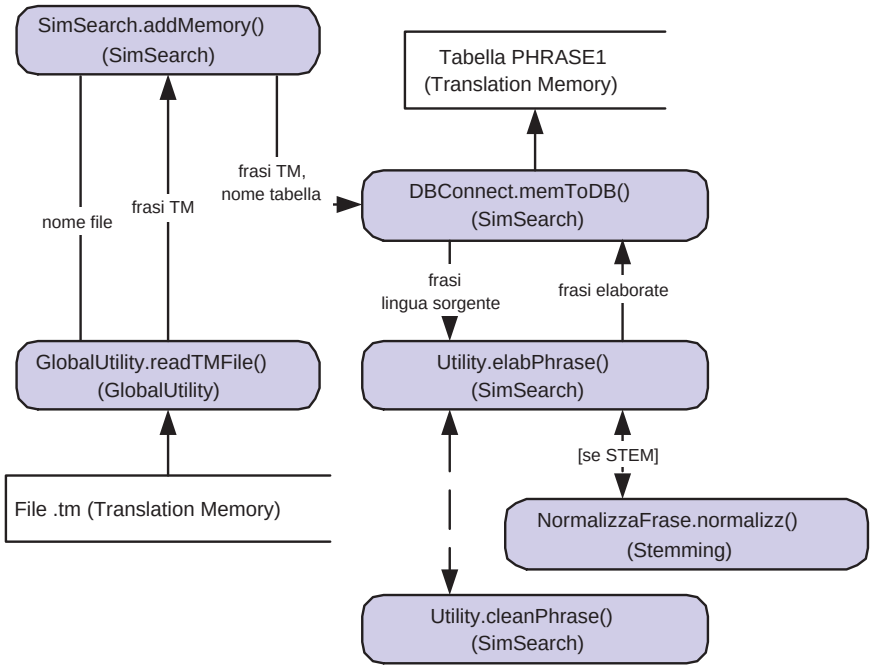


Figura 5.20: Aggiunta di dati alla TM: data flow diagram (parte 1)

In figura 5.21 viene mostrata l'ulteriore operazione svolta: l'estrazione e la registrazione dei due set di q-grammi (uno di dimensione q, registrato nelle tabelle $QPhrase$, ed uno di dimensione minore $qSub$, registrato nelle tabelle $QSPhrase$), richiesti dalle operazioni di ricerca che verranno svolte sulla TM. Vengono utilizzate le seguenti funzioni:

- *qGramVector()*, applicata alla frase estesa con i caratteri iniziali e finali, restituisce il vettore di q-grammi risultanti;
- *fillQtable()* provvede ad interfacciarsi con il DBMS e a registrare i q-grammi nelle opportune tabelle.

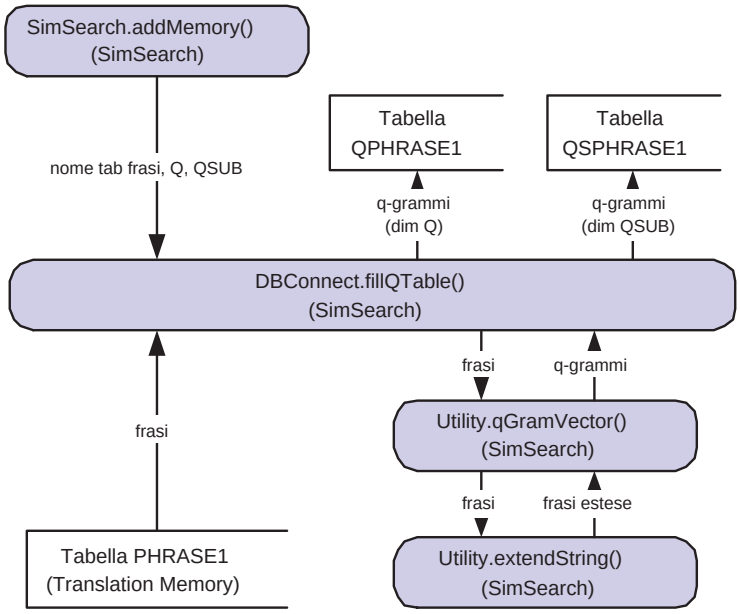


Figura 5.21: Aggiunta di dati alla TM: data flow diagram (parte 2)

Capitolo 6

Le prove sperimentali e i risultati ottenuti

In questo capitolo vengono mostrati i risultati ottenuti utilizzando gli approcci fin qui descritti; in particolare, si analizzerà tanto l'efficacia quanto l'efficienza delle varie funzionalità offerte da EXTRA, sia per quanto riguarda gli algoritmi di allineamento, sia per la ricerca di similarità nel suo insieme.

Per compiere questa analisi sono stati svolti un notevole numero di test e prove sperimentali, appositamente studiati per mettere alla prova il sistema nel modo più completo possibile, mostrandone il comportamento al variare dei principali parametri; di questi test verranno riportati i più significativi. Inoltre, si confronteranno le prestazioni ottenute dal sistema con quelle offerte dagli approcci tradizionali, come descritti nella prima parte di questo lavoro.

Per i test riguardanti l'efficienza, si ricorda per la precisione che i tempi riportati sono riferiti ad un computer basato su Pentium III 1000Mhz con 256 Mb di memoria ed hard disk EIDE da 20 Gb.

6.1 Le collezioni usate nei test

Essendo i sistemi EBMT un campo di ricerca relativamente nuovo, non esistono criteri o test precisi per verificarne le prestazioni. In ambito di Information Retrieval classico esistono collezioni di documenti (e relative query) di riferimento, unanimamente riconosciute ed utilizzate dai ricercatori per verificare l'efficacia e l'efficienza di un algoritmo. Nel nostro caso è stato evidente fin da subito che queste collezioni non potevano essere utilizzate: esse sono costruite per verificare

quali documenti vengono reperiti da un sistema IR data una serie parole chiave descrittive, problema ben diverso da quello della ricerca di similarità tra frasi.

Per effettuare le prove sperimentali sul sistema sono state pertanto utilizzate una serie di collezioni create ad hoc:

- *Collezione "DPaint"*: si tratta di due versioni successive dello stesso manuale tecnico di un prodotto software. È costituita da 1497 frasi di riferimento e da 419 frasi di query.

- *Collezione "Logos"*: si tratta di una Translation Memory completa relativa ai manuali tecnici di questa ditta. Le frasi di riferimento sono in questo caso 34550, complete della relativa traduzione, e 421 le frasi di query.

La collezione *DPaint* è stata utilizzata più che altro per analizzare il sistema qualitativamente, dal punto di vista dell'efficacia dei suggerimenti. L'idea alla base di questa collezione è quella di testare il sistema in una delle situazioni più comuni per la traduzione EBMT: dover tradurre una nuova versione di un manuale tecnico disponendo delle traduzioni di una versione precedente, in cui presumibilmente sarà possibile riscontrare notevoli similarità. Per rendere più interessanti le prove, sono stati scelti manuali dalla versione 3 e dalla versione 5 di questo software: versioni non consecutive, proprio per ridurre le similarità banali dovute alla presenza di frasi rimaste invariate. In questo modo è stato possibile mettere maggiormente alla prova gli algoritmi avanzati di ricerca di sottoparti simili.

La collezione *Logos* è invece stata fornita da LOGOS così come da loro utilizzata: essa rappresenta tutta la translation memory relativa ad un particolare cliente e permette di testare il sistema con i volumi di lavoro utilizzati effettivamente dai traduttori professionisti. Pertanto, questa collezione è stata utilizzata soprattutto per testare il sistema dal punto di vista dell'efficienza; inoltre, è stata utile per verificarne il comportamento in uno scenario più consolidato rispetto al precedente, in cui il numero e la ripetitività delle frasi nella Translation Memory erano molto inferiori. Infine, a differenza della collezione *DPaint*, questa collezione contiene le traduzioni delle frasi di riferimento: in questo modo è stato possibile testare anche l'efficacia degli allineamenti nel fornire i suggerimenti nella lingua destinazione.

Nelle figure 6.1 e 6.2 sono mostrate per completezza le distribuzioni delle lunghezze delle frasi delle due collezioni, così come mostrate dall'apposita funzione

descritta nella sezione 4.7: in particolare è possibile notare che il picco del grafico relativo alla collezione *Logos* è centrato su lunghezze inferiori, in quanto in essa compaiono, oltre alle frasi vere e proprie, anche le terminologie utilizzate con le relative traduzioni.

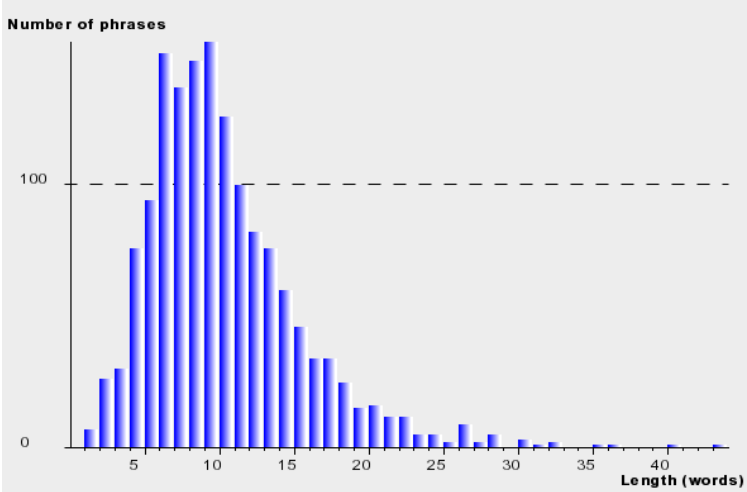


Figura 6.1: Distribuzione della collezione DPaint

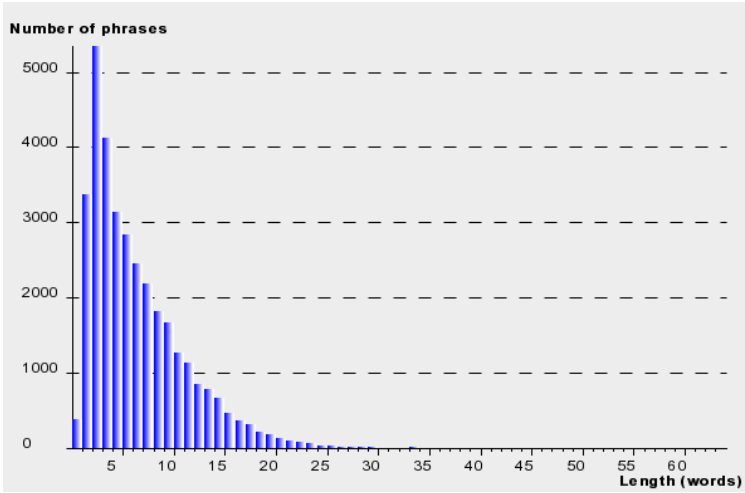


Figura 6.2: Distribuzione della collezione Logos

Oltre alle collezioni citate, poiché la collezione *DPaint* contiene frasi in una sola lingua, mentre la collezione *Logos* è bilingue ma con frasi già correttamente allineate dai traduttori stessi, sono stati utilizzati dei documenti aggiuntivi per testare più accuratamente gli algoritmi di allineamento, come si vedrà nelle sezioni successive.

6.2 Allineamento delle frasi

Per mostrare l'efficacia degli algoritmi di allineamento delle frasi si considerano dei paragrafi bilingui tratti dai dibattiti parlamentari Canadian Hansards, in inglese e francese. I paragrafi di partenza sono mostrati nelle figure 6.3 e 6.4.

Inglese	Francese
According to our survey, 1988 sales of mineral water and soft drinks were much higher than in 1987, reflecting the growing popularity of these products. Cola drink manufacturers in particular achieved above-average growth rates. The higher turnover was largely due to an increase in the sales volume. Employment and investment levels also climbed. Following a two-year transitional period, the new Foodstuffs Ordinance for Mineral Water came into effect on April 1, 1988. Specifically, it contains more stringent requirements regarding quality consistency and purity guarantees.	Quant aux eaux minérales et aux limonades, elles rencontrent toujours plus d'adeptes. En effet, notre sondage fait ressortir des ventes nettement supérieures à celles de 1987, pour les boissons à base de cola notamment. La progression des chiffres d'affaires résulte en grande partie de l'accroissement du volume des ventes. L'emploi et les investissements ont également augmenté. La nouvelle ordonnance fédérale sur les denrées alimentaires concernant entre autres les eaux minérales, entrée en vigueur le 1er avril 1988 après une période transitoire de deux ans, exige surtout une plus grande constance dans la qualité et une garantie de la pureté.

Figura 6.3: Allineamento frasi: i paragrafi di partenza (esempio 1)

Inglese	Francese
The crisis our farmers are in right now will affect all of us at a certain point in time. We are all consumers and we all need a strong and healthy agricultural sector. I am glad that the Hon Member for Algoma mentioned figures in his remarks. Otherwise, the Government might have eluded the problem once again. The Hon Member for Algoma suggested Tuesday night that the Government had to take a clear position and make a commitment to assist our farmers before it is too late.	La crise que vivent en ce moment nos agriculteurs se répercutera sur tous et chacun de nous à un certain moment. Nous sommes des consommateurs. Nous avons tous besoin d'une agriculture saine et forte. Heureusement que le député d'Algoma a mentionné des chiffres dans ses remarques, sans cela ce gouvernement s'en serait sorti en douce encore une fois. Le député d'Algoma suggérait mardi soir qu'il fallait que le gouvernement se prononce clairement et s'engage à aider nos agriculteurs avant qu'il ne soit trop tard.

Figura 6.4: Allineamento frasi: i paragrafi di partenza (esempio 2)

Forniti i paragrafi in input, in essi vengono automaticamente riconosciute le varie frasi e quindi l'allineamento delle frasi fornisce in output il risultato, mostrato nelle figure 6.5 e 6.6.

Inglese	Francese	Match	Dist
According to our survey, 1988 sales of mineral water and soft drinks were much higher than in 1987, reflecting the growing popularity of these products.	Quant aux eaux minérales et aux limonades, elles rencontrent toujours plus d'adeptes.	2 :2	460
Cola drink manufacturers in particular achieved above-average growth rates.	En effet, notre sondage fait ressortir des ventes nettement supérieures à celles de 1987, pour les boissons à base de cola notamment.		
The higher turnover was largely due to an increase in the sales volume.	La progression des chiffres d'affaires résulte en grande partie de l'accroissement du volume des ventes.	1 :1	173
Employment and investment levels also climbed.	L'emploi et les investissements ont également augmenté.	1 :1	46
Following a two-year transitional period, the new Foodstuffs Ordinance for Mineral Water came into effect on April 1, 1988.	La nouvelle ordonnance fédérale sur les denrées alimentaires concernant entre autres les eaux minérales, entrée en vigueur le 1er avril 1988 après une période transitoire de deux ans, exige surtout une plus grande constance dans la qualité et une garantie de la pureté.	2 :1	340
Specifically, it contains more stringent requirements regarding quality consistency and purity guarantees.			

Figura 6.5: Allineamento frasi: i risultati (esempio 1)

Inglese	Francese	Match	Dist
The crisis our farmers are in right now will affect all of us at a certain point in time.	La crise que vivent en ce moment nos agriculteurs se répercutera sur tous et chacun de nous à un certain moment.	1 :1	97
We are all consumers and we all need a strong and healthy agricultural sector.	Nous sommes des consommateurs.	1 :2	260
	Nous avons tous besoin d'une agriculture saine et forte.		
I am glad that the Hon Member for Algoma mentioned figures in his remarks.	Heureusement que le député d'Algoma a mentionné des chiffres dans ses remarques, sans cela ce gouvernement s'en serait sorti en douce encore une fois.	2 :1	255
Otherwise, the Government might have eluded the problem once again.			
The Hon Member for Algoma suggested Tuesday night that the Government had to take a clear position and make a commitment to assist our farmers before it is too late.	Le député d'Algoma suggérait mardi soir qu'il fallait que le gouvernement se prononce clairement et s'engage à aider nos agriculteurs avant qu'il ne soit trop tard.	1 :1	2

Figura 6.6: Allineamento frasi: i risultati (esempio 2)

Come si vede le frasi vengono correttamente messe in corrispondenza, individuando automaticamente e con successo anche le situazioni più complesse, quali match 2:2 o 2:1, in cui non vi è un semplice legame tra le frasi.

Per sottolineare come i risultati ottenuti siano del tutto indipendenti dalle lingue utilizzate, in figura 6.7 è mostrato l'allineamento per lo stesso paragrafo, questa volta tra la versione inglese e quella italiana.

Inglese	Italiano	Match	Dist
The crisis our farmers are in right now will affect all of us at a certain point in time.	La crisi che in questo momento stanno vivendo i nostri agricoltori prima o poi si ripercuoterà su ciascuno di noi.	1 :1	107
We are all consumers and we all need a strong and healthy agricultural sector.	Siamo dei consumatori.	1 :2	241
	Abbiamo tutti bisogno di un'agricoltura sana e forte.		
I am glad that the Hon Member for Algoma mentioned figures in his remarks.	Siamo felici che il deputato di Algoma abbia mostrato dei dati nella sua analisi, altrimenti il governo l'avrebbe fatta franca ancora una volta.	2 :1	237
Otherwise, the Government might have eluded the problem once again.			
The Hon Member for Algoma suggested Tuesday night that the Government had to take a clear position and make a commitment to assist our farmers before it is too late.	Martedi sera, il deputato di Algoma ha suggerito la necessità che il governo si pronunci chiaramente ed inizi ad aiutare noi agricoltori prima che sia troppo tardi.	1 :1	2

Figura 6.7: Allineamento frasi: i risultati (esempio 2 bis)

L'efficacia di questo algoritmo è indubbiamente notevole e gli allineamenti forniti sono rivelati corretti in oltre il 95% dei paragrafi allineati. Per concludere, un cenno anche all'efficienza, altrettanto buona: data la sua relativa semplicità, esso è in grado di allineare oltre 1500 frasi in poco più di 3 secondi.

6.3 Allineamento delle parole

Vengono di seguito riportati i risultati di alcuni allineamenti tra parole, in questo caso tra frasi in inglese ed italiano della collezione *Logos* e di altre fonti aggiuntive. Di questi allineamenti viene mostrata la funzione di allineamento come ricavata dall'algoritmo, con evidenziati in particolare i punti di corrispondenza trovati: con un simbolo pieno vengono mostrati i match esatti tra gli elementi (parole, elementi di punteggiatura, e così via), con uno vuoto i match ricavati tramite gli algoritmi LCS di somiglianza.

In figura 6.8 si vede come l'algoritmo è in grado di ricostruire un allineamento di notevole precisione tra due frasi che presentano alcune inversioni nella strut-

tura (ad esempio “search time” e “tempo di ricerca”, espressioni con un ordine delle parole invertito) e con nessun elemento identico (a meno del punto finale). Tutto questo grazie all’algoritmo di LCS, che permette di cogliere le notevoli similarità tra coppie di parole quali “observations” e “osservazioni”, “reduce” e “ridurre”, e così via.

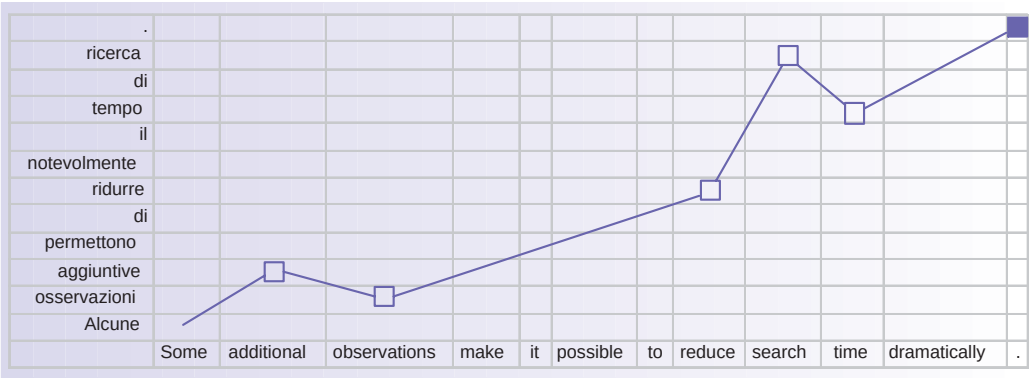


Figura 6.8: Allineamento parole: i risultati (esempio 1)

In figura 6.9 è mostrato un altro esempio: in questo caso la corrispondenza tra le frasi è più lineare, ma le parole uguali o simili sono quasi inesistenti.

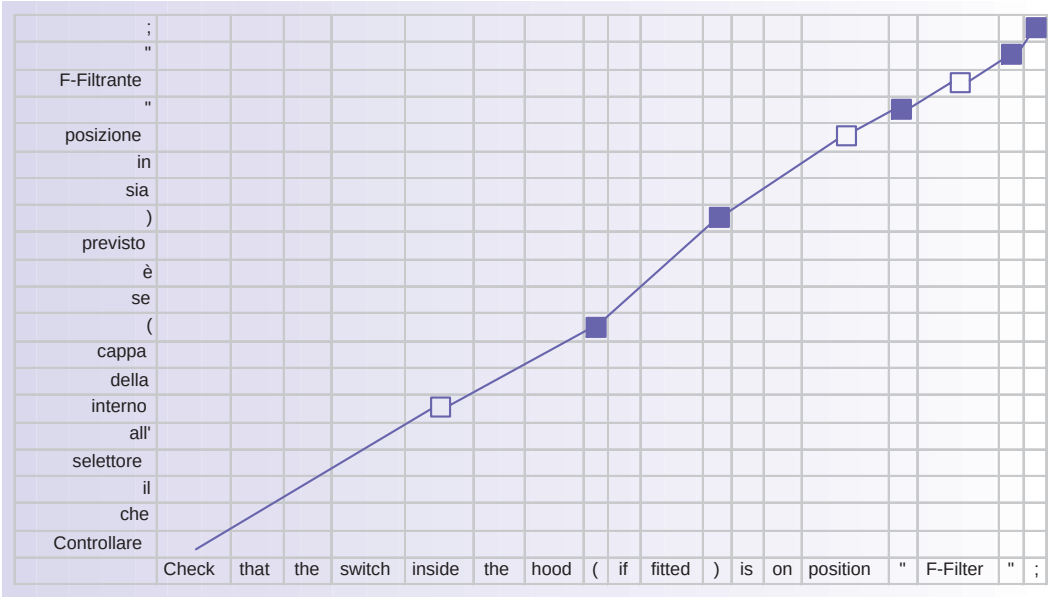


Figura 6.9: Allineamento parole: i risultati (esempio 2)

L’algoritmo è qui in grado di risalire al corretto allineamento soprattutto grazie alla funzione di individuazione ed accoppiamento della punteggiatura, con

le parentesi e le virgolette che forniscono punti di corrispondenza praticamente certi.

Nelle figure 6.10 e 6.11 sono mostrate frasi dalla struttura abbastanza particolare, con pesanti inversioni nell'ordine delle parole. Anche in questi casi complessi, l'allineamento produce risultati incoraggianti.

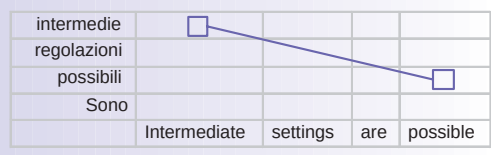


Figura 6.10: Allineamento parole: i risultati (esempio 4)

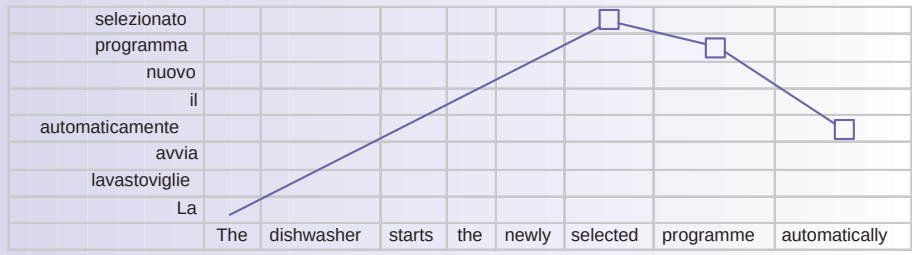


Figura 6.11: Allineamento parole: i risultati (esempio 5)

Per sottolineare la sostanziale indipendenza dai linguaggi utilizzati, in figura 6.12 è mostrato un esempio di allineamento tra due frasi in polacco ed inglese.

Da notare che:

- la sigla "ISSO-123" è stata trovata identica tra le due frasi e costituisce pertanto un punto saldo dell'allineamento;
- le parole "information" e "informacje", nonché "products" e "produkty" sono messe in corrispondenza grazie all'algoritmo di somiglianza (LCS), efficace anche per queste lingue;
- il termine "standard" e le sue varianti compaiono due volte in ciascuna frase: grazie alla funzione di smorzamento di distanza vengono messe in corrispondenza nel modo più opportuno.

L'efficacia raggiunta dall'algoritmo di allineamento delle parole è di tutto rispetto, fornendo questo, come si è visto, allineamenti corretti o comunque

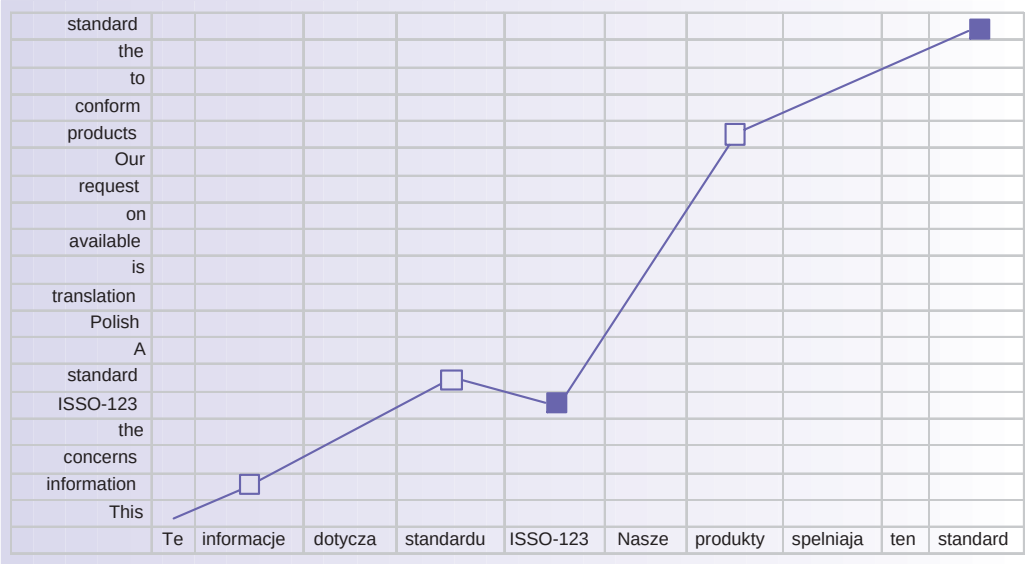


Figura 6.12: Allineamento parole: i risultati (esempio 6)

sufficientemente precisi per supportare correttamente la successiva estrazione delle parti simili (vedi in proposito la sezione successiva). Delle oltre 35000 frasi allineate per la collezione *Logos* solo una percentuale molto bassa contenevano delle corrispondenze inesatte; inoltre, bisogna tener conto che nella maggior parte dei casi, è possibile ovviare agli eventuali errori di allineamento modificando in modo opportuno i vari parametri a disposizione, che permettono di adattare al meglio l'algoritmo ai documenti e alle lingue su cui si opera.

Per quanto riguarda l'efficienza, anche questo algoritmo è molto veloce: le oltre 35000 frasi della collezione vengono allineate in poco meno di un minuto e mezzo.

6.4 Ricerca di similarità

In questa sezione vengono analizzate le prestazioni della principale funzione del programma, la ricerca di similarità. In particolare si esamineranno:

- l'*efficacia*, con la copertura raggiunta sulle collezioni e la qualità dei suggerimenti proposti;
- l'*efficienza* delle varie modalità disponibili;

- gli effetti delle variazioni sui principali *parametri* disponibili, nonché dei *filtri* e degli *indici* utilizzati.

6.4.1 Efficacia

Copertura

Un aspetto fondamentale dell'efficacia della ricerca di similarità è la copertura che gli algoritmi di ricerca sono in grado di fornire sulle varie collezioni. Come si vede dal grafico di figura 6.13, essi sono in grado di fornire suggerimenti per la maggior parte delle frasi cercate.

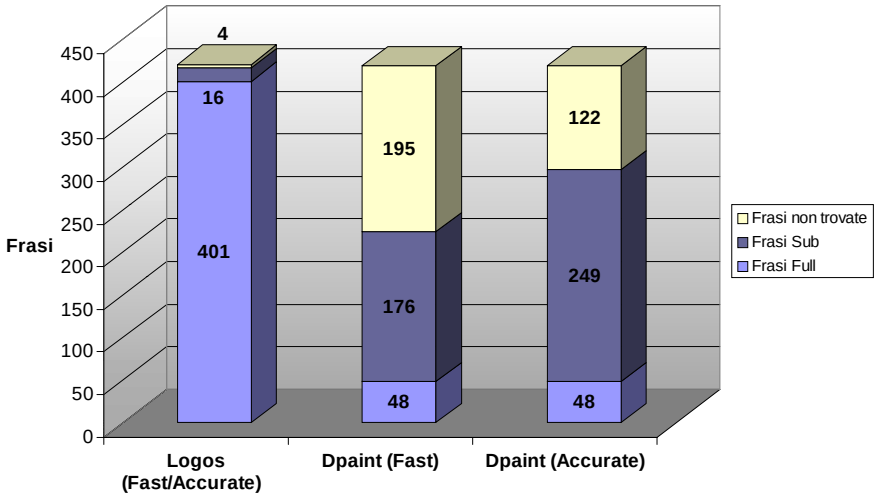


Figura 6.13: Copertura ottenuta sulle collezioni

Per la collezione *Logos*, che dispone di una translation memory più ricca e consolidata, la maggior parte dei suggerimenti è costituita da frasi intere simili. In questo caso, la copertura rimane invariata (con oltre il 99% di frasi trovate) passando da una ricerca veloce ad una accurata e i dati presentati per essa riguardano entrambe queste modalità di ricerca.

Utilizzando invece la collezione *DPaint*, più piccola e pertanto meno ottimale dal punto di vista della quantità di frasi da cui attingere i suggerimenti, si vede che gli algoritmi di ricerca di sottoparti assumono un ruolo molto più importante, permettendo di raggiungere un'ottima copertura anche in questa difficile situazione, soprattutto nel caso di ricerca accurata (71% di frasi trovate).

I suggerimenti forniti

Vengono ora riportati degli esempi riguardanti la ricerca di similarità: si tratta di esempi di pretraduzione delle collezioni *Logos* e *DPaint*, che ben rappresentano l'efficacia dei suggerimenti di traduzione forniti.

Come si è visto, il programma individua prima di tutto le frasi della translation memory che risultano sufficientemente simili nella loro interezza a quelle cercate. Un esempio di *fullmatch* è il seguente:

> **Frase cercata:**
> Position the 4 clips (D) as shown and at the specified dimensions.
> *(Position clip show specify dimension)*
>
> **Frase intera simile (Dist 20%):**
> Position the 4 clips (A) as shown and at the specified distance.
> *(Position clip show specify distance)*
>
> **Traduzione suggerita:**
> Posizionare le 4 mollette (A) come indicato e alla distanza prevista.

Tra parentesi sono riportate le frasi elaborate mediante stemming, su cui avviene la ricerca e il calcolo di distanza: in questo caso la frase trovata è distante il 20% rispetto alla frase cercata (edit distance di 1 sulle 5 parole della frase).

Ancora più significativi sono i seguenti esempi di *submatch*, in cui il programma propone suggerimenti parziali che permettono di facilitare notevolmente la traduzione anche di quelle frasi per cui non è presente nella translation memory nessuna frase simile per intero. Per non appesantire troppo gli esempi, non vengono ulteriormente riportate le frasi normalizzate e i punteggi di similarità, comunque presenti nei file di output generati. Vengono invece evidenziate in *corsivo* le parti riconosciute come simili.

> **Frase cercata:**
> *Insert the power supply cable into the cable clamp and connect the conductors to the mounting plate as indicated in the diagram printed* on the cover.
>
> **Frase con parte simile:**
> Strip approx. 10 mm of sheath from the wires. First insert the cable (max. 14 mm diameter) through the opening in the cover. Then

insert the power supply cable into the cable clamp and connect the wires to the terminal block as indicated in the diagram printed on the bottom of the cooktop.

Traduzione completa:
Rimuovere ca. 10 mm di guaina dai fili. Inserire dapprima il cavo (diametro max 14 mm) attraverso l' apposita apertura nel coperchio. *Inserire quindi il cavo alimentatore nel morsetto serrafilo e collegare i fili al blocco terminale come indicato nel diagramma stampato* sulla base del piano di cottura.

Traduzione suggerita:
Inserire quindi il cavo alimentatore nel morsetto serrafilo e collegare i fili al blocco terminale come indicato nel diagramma stampato

La similarità viene riconosciuta sulle frasi nella lingua sorgente; individuata la parte simile, il programma accede ai dati di allineamento ed è così in grado di estrarre dalla traduzione della frase completa la parte interessata. Si ricorda che per gli esempi tratti dalla collezione *Logos*, come quello appena mostrato, la segmentazione è stata indicata dai traduttori stessi e non è ricavata dagli algoritmi di EXTRA: gli elementi coinvolti non sempre sono costituiti da singole frasi, ma come si vede questo non condiziona la qualità della pretraduzione. Un altro esempio è il seguente:

Frase cercata:
On completion of *electrical connections, fit the cooktop in place from the top* and *secure it by means of the clips* as shown.

Frase con parte simile 1:
After the *electrical connection, fit the hob from the top* and hook it to the support springs, according to the illustration.

Traduzione completa 1:
Dopo aver eseguito il *collegamento elettrico, montare il piano cottura dall'alto* e agganciarlo alle molle di supporto come da figura.

Traduzione suggerita 1:
collegamento elettrico, montare il piano cottura dall'alto

Frase con parte simile 2:
Secure it by means of the clips.

Traduzione suggerita 2:
Fissare definitivamente per mezzo dei ganci.

In questo caso, per una stessa frase vengono forniti più suggerimenti parziali, che insieme la coprono quasi completamente. Altri esempi verranno mostrati nella sezione 6.5, dedicata al confronto con altri approcci.

6.4.2 Efficienza

I tempi nelle varie modalità di ricerca

In figura 6.14 vengono mostrate le prestazioni relative alle diverse modalità di ricerca disponibili. In particolare, vengono riportati i tempi delle varie query di ricerca relativi alla collezione *Logos*, la più indicata per test di questo tipo date le sue ragguardevoli dimensioni; i tempi di inserimento delle frasi da pretradurre (complete dei relativi q-grammi) nelle tabelle del DMBS sono uguali per tutte le modalità e non vengono riportati in quanto trascurabili rispetto al tempo di ricerca. La modalità di ricerca di sottoparti ad un solo passaggio non è stata considerata in quanto, disattivandone i filtri per garantire una efficacia elevata quanto nelle altre modalità, si è rivelata da subito nettamente meno efficiente.

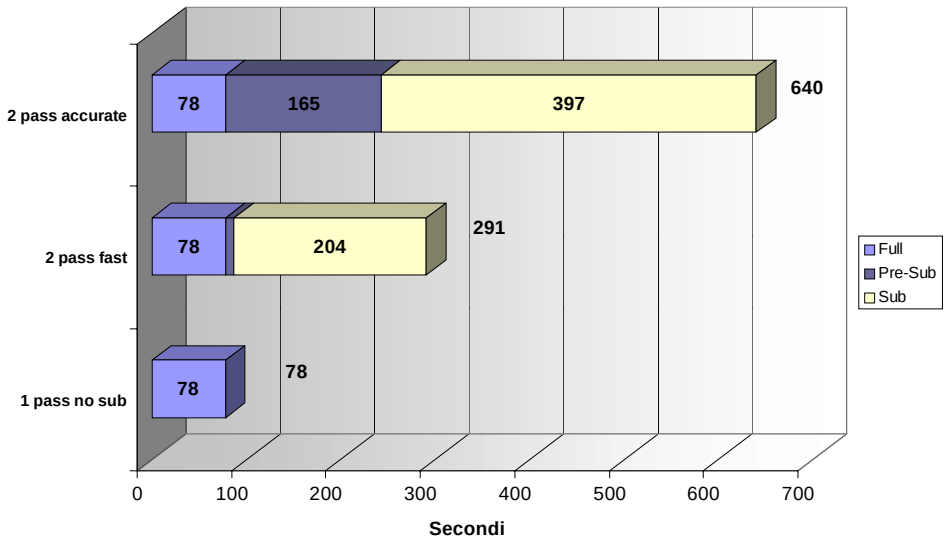


Figura 6.14: Confronto tra i tempi nelle varie modalità di ricerca

Come si vede, grazie sopratutto all'uso dei filtri e degli indici mostrati nel Capitolo 3, i tempi di ricerca sono notevolmente contenuti, soprattutto tenendo

conto che la ricerca riguarda in questo caso 421 frasi da pretradurre con una translation memory di oltre 34000 frasi: poco più di un minuto per ricercare tutte le possibili somiglianze tra frasi intere, circa quattro minuti per ricercare anche le somiglianze tra sottoparti nel modo veloce, meno di dieci minuti per eseguire la ricerca nel modo accurato, evidenziando tutti i risultati esistenti. In particolare è interessante notare come anche nel modo più accurato i tempi medi totali per frase cercata siano di circa 1.5 secondi: un risultato molto incoraggiante e che riesce a rimanere ben al di sotto del tempo di ricerca medio dei normali algoritmi, che pure si limitano alla ricerca di frasi intere.

Scalabilità

Sempre sulla collezione *Logos* sono stati eseguiti test di scalabilità. In questo caso, per ciascun carico di lavoro (25%, 50% e 100%), sono stati generati 10 file da pretradurre contenenti frasi diverse tra loro ed estratte casualmente tra le 421 a disposizione per le query: dei tempi ottenuti in ciascuna delle tre query è stata quindi eseguita la media. I risultati sono mostrati in figura 6.15.

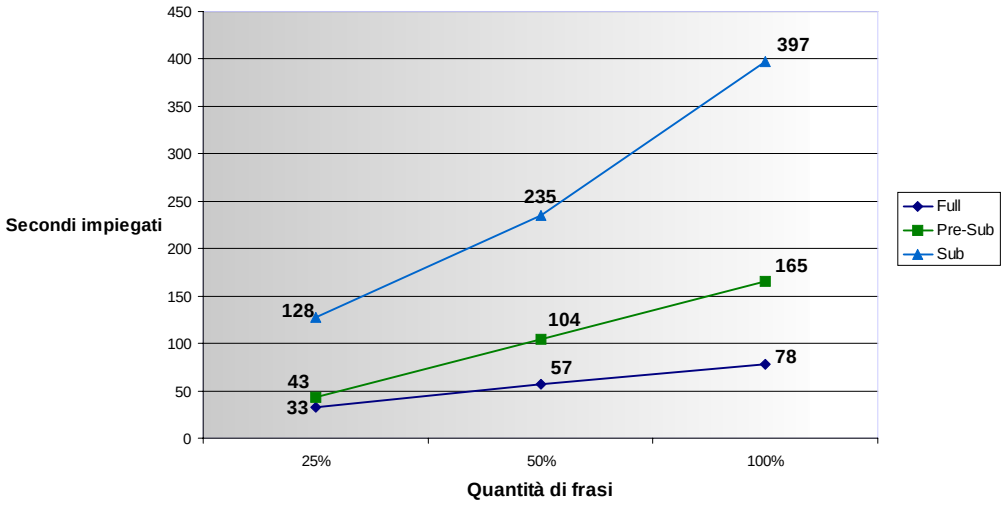


Figura 6.15: Test di scalabilità

Come messo in evidenza dai test:

- i tempi della query per le frasi intere presentano un andamento meno che lineare all'aumentare del numero delle frasi da pretradurre;

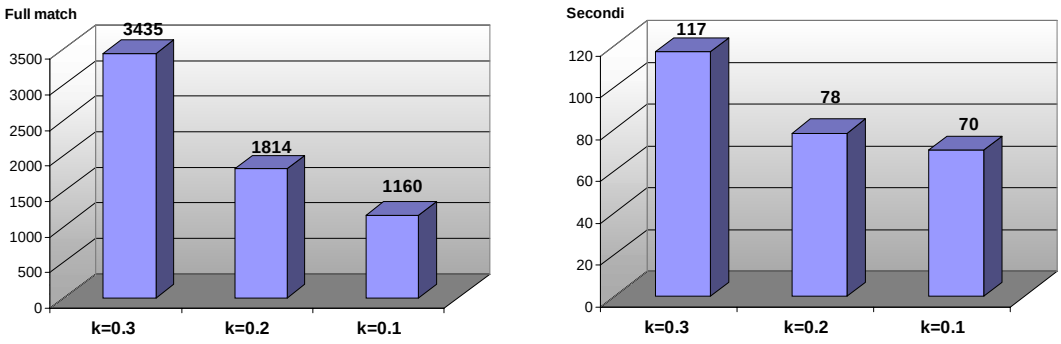
- le query per la preparazione e la ricerca delle sottoparti presentano un andamento approssimativamente lineare. La query di ricerca delle sottoparti è quella che richiede più tempo all'aumento del carico di lavoro, fatto comprensibile se si pensa alla notevole complessità delle operazioni ad essa richieste.

6.4.3 Variazioni dei parametri

I risultati mostrati fino ad ora sono riferiti ai valori di default dei parametri k e q, rispettivamente fissati a 0.2 (errore massimo ammesso del 20%) e 3 (utilizzo di tri-grammi).

Gli effetti della variazione di questi parametri sono mostrati dai test di figura 6.16 e 6.17, relativi alla collezione *Logos* e alla ricerca di frasi intere: come ci si può aspettare, all'aumentare dell'errore massimo ammesso (k) aumentano sia i tempi di esecuzione sia il numero di match trovati e suggeriti, mentre la variazione di q ha effetto solo ed esclusivamente sui tempi.

La figura 6.17 evidenzia tra l'altro come i filtri basati su tri-grammi risultino i più efficienti.



(a) Numero di frasi trovate

(b) Tempo impiegato

Figura 6.16: Variazioni di k

6.4.4 Effetto di filtri e indici

In tutti i test visti sono stati utilizzati i filtri e gli indici mostrati nel capitolo 3 e studiati appositamente per permettere una buona efficienza generale. Per

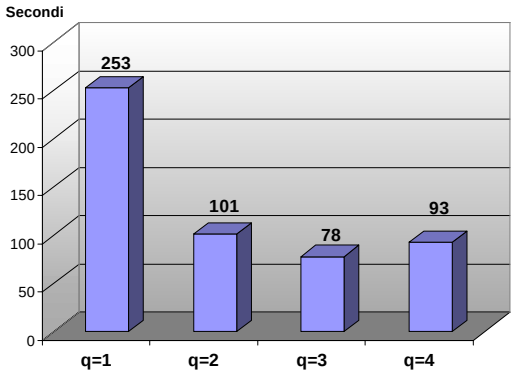


Figura 6.17: Variazioni di q: tempo impiegato

meglio capire il loro effetto e quanto essi migliorino le prestazioni, vengono qui mostrati i risultati di test svolti appositamente con questo scopo.

In figura 6.18 sono mostrati i tempi di pretraduzione (frasi intere) con filtri ed indici alternativamente disattivati. I test utilizzati sono i seguenti:

- Test1: pretraduzione di 421 frasi di query con una translation memory di 34550 frasi;
- Test2: pretraduzione di 50 frasi con una translation memory di 61420 frasi.

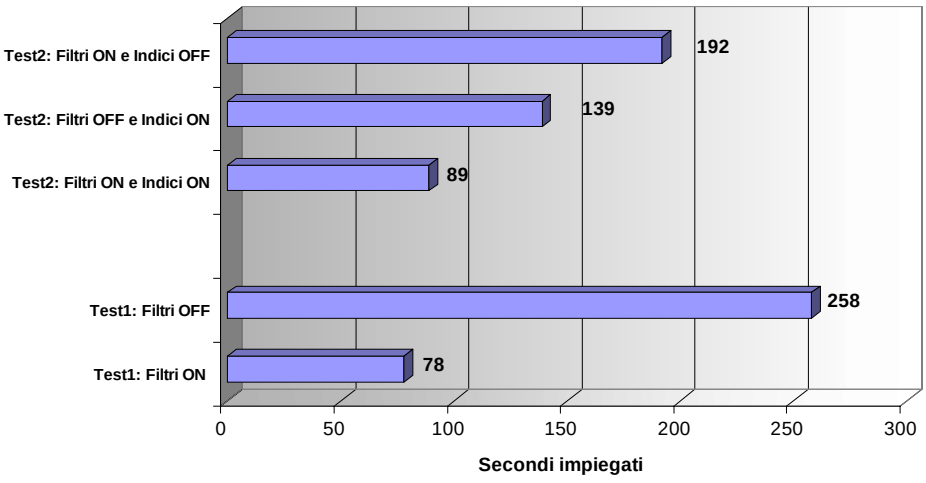


Figura 6.18: Effetti sui tempi di filtri e indici

Come si vede il beneficio è più che evidente: attivare tutti i filtri riduce il tempo di risposta da 258 a 78 secondi. La presenza degli indici migliora

ulteriormente le prestazioni, come mostrato dal Test2. In particolare, le riduzioni dei tempi risultano comprese tra il 35% e 70%.

6.5 Confronto con altri approcci

Per avere un'idea dei risultati ottenibili con gli approcci tradizionali descritti nella prima parte, e di come essi si rapportino a quelli ottenuti dal nostro sistema, vengono fatti alcuni esempi in questa sezione.

6.5.1 Approcci IR classici

Sono stati innanzitutto analizzati i risultati ottenibili sulle collezioni di riferimento (in particolare la collezione *DPaint*) mediante l'utilizzo di algoritmi più strettamente legati all'ambito dell'Information Retrieval (capitolo 1): in particolare la ricerca basata sui modelli vettoriali classici (descritti nella sezione 1.3.3) e su query per DBMS con estensioni per Text Retrieval, quali quelle sviluppate da F. Gavioli [13] (sezione 1.8.2).

I limiti principali di questi approcci sono risultati evidenti: la ricerca di similarità è limitata alle sole frasi intere, permettendo in questo modo un'inferiore copertura delle collezioni. In particolare è stato possibile confrontare dettagliatamente la copertura ottenibile da EXTRA sulla collezione *DPaint* con quella delle query di ricerca basate su estensioni per Text Retrieval, descritte nella sezione 1.8.2.

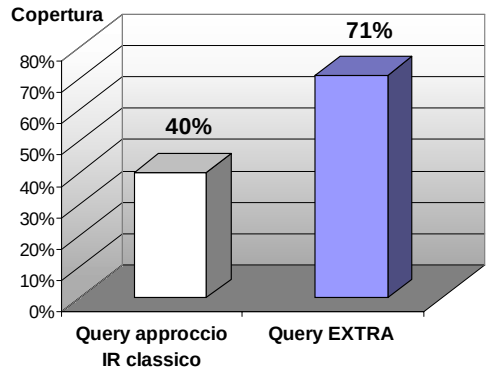


Figura 6.19: Confronto di copertura

In figura 6.19 si può vedere che con tali algoritmi non viene superato il 40% di

frasi trovate, contro il 71% ottenuto dagli algoritmi utilizzati in EXTRA (valori standard dei parametri).

Per fare degli esempi, le seguenti frasi trovate da EXTRA, effettivamente utili, non vengono individuate da approcci classici:

Frase cercata:
If you're an *experienced computer graphics artist, you might want to* move directly to About this Manual.

Frase con parte simile (EXTRA):
If you have some *experience with computer graphics software, you may want to* use this chapter to learn how DPaint handles features you may have encountered in other programs.

Frase cercata: *These chapters describe the fundamentals* of the program and introduce some advanced techniques.

Frase con parte simile 1 (EXTRA): *In this chapter we examine the fundamental* elements that make up DPaint.

Frase con parte simile 2 (EXTRA): *That chapter introduces you to the fundamentals* of perspective in DPaint III.

Inoltre, la metrica di similarità utilizzata tiene fondamentalmente conto solo del numero delle parole uguali presenti nelle frasi, ignorando un'informazione fondamentale quale la posizione e soprattutto l'ordine delle parole. La diretta conseguenza è che spesso vengono proposti suggerimenti che sono di ben poca utilità per il traduttore. Un esempio è fornito dalle seguenti frasi:

Frase cercata:
But you don't have to memorize the *manual* to master the *program*.

Frase suggerita (modello classico):
This file contains information about the *program* that was not available before the *manual* went to press.

Esse contengono parole in comune (in *corsivo*) ma è evidente che non presentano somiglianze tali da poter essere considerate effettivamente simili. Ancora:

Frase cercata:
They *both* mark how to sections that *present* something you should do to *understand* a program feature or function.

Frase suggerita 1 (modello classico):
This removes them *both* if they are *present*.

Frase suggerita 2 (modello classico):
Understand that each click of the mouse toggles the locking *function* on and off, just as it does in *both* requesters.

Il suggerimento individuato da EXTRA è invece più mirato e, pur non coprendo l'intera frase, risulta comunque utile al traduttore, grazie anche alla corretta estrazione delle parti interessate in lingua sorgente e destinazione:

Frase simile (EXTRA):
We've worked hard to make this explanation straightforward, but due to the complexity of Perspective itself, you should ensure to *understand other program features*.

Traduzione suggerita (EXTRA):
comprendere le altre caratteristiche del programma.

Dal punto di vista dell'efficienza è stato eseguito un confronto tra il tempo impiegato da EXTRA nell'eseguire la pretraduzione della collezione DPaint e quello impiegato dalle query di ricerca classiche di Text Retrieval.

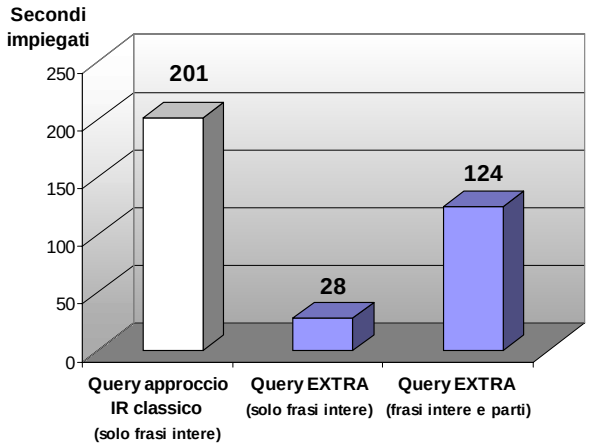


Figura 6.20: Confronto di efficienza

Ancora una volta i filtri e le ottimizzazioni apportate alle query del sistema permettono risultati migliori, consentendo di eseguire ricerche più complesse (riguardanti non solo le frasi intere) in tempi comunque complessivamente inferiori.

6.5.2 Altri sistemi MT

Innanzitutto, per dimostrare anche sperimentalmente come i sistemi HAMT per la traduzione completamente automatica non siano ancora sufficientemente maturi per essere di vera utilità, sono stati effettuati dei test di traduzione delle frasi delle collezioni con uno dei più diffusi di questi strumenti. I risultati, riassunti nell'esempio che segue, confermano come, all'attuale stato della tecnologia, sistemi EBMT come EXTRA siano di gran lunga più promettenti ed efficaci.

> **Frase cercata:**
> Insert the power supply cable into the cable clamp and connect the conductors to the mounting plate as indicated in the diagram printed on the cover.
>
> **Traduzione automatica HAMT:**
> Inserisca il cavo di approvvigionamento di potere nel morsetto del cavo e connetta i bigliettai al piatto che monta come indicò nel diagramma stampato sulla coperta.
>
> **Traduzione parziale suggerita (EXTRA):**
> Inserire quindi il cavo alimentatore nel morsetto serrafilo e collegare i fili al blocco terminale come indicato nel diagramma stampato sul coperchio.

Per quanto riguarda invece gli ambienti EBMT commerciali descritti nel capitolo 2 valgono considerazioni praticamente analoghe a quelle viste per gli approcci IR classici: la mancanza della ricerca relativa alle sottoparti è una limitazione fondamentale anche in questo caso. Di conseguenza non vengono riportati ulteriori esempi, che risulterebbero analoghi a quelli mostrati nella sezione precedente: dai test svolti, tali sistemi permettono risultati di buona qualità, ma solo in presenza di Translation Memory di notevoli dimensioni e ben consolidate, in cui è altamente probabile trovare frasi identiche (o quasi) a quelle cercate. In questi casi, essi sono anche in grado di suggerire frasi tradotte già modificate,

sostituendo le parole riconosciute come diverse con quelle delle nuove frasi da tradurre.

Poiché però è piuttosto ottimistico presupporre che per ogni frase da tradurre ne sia presente una sostanzialmente identica nella Translation Memory, la scarsa flessibilità degli algoritmi di ricerca limita in molti casi l'effettiva utilità di questi ambienti, sia come copertura sia come qualità dei suggerimenti.

Conclusioni e sviluppi futuri

In questo lavoro sono stati raggiunti i seguenti risultati:

- è stata definita una *metrica di similarità* tra frasi, basata su un inedito utilizzo del concetto di Edit Distance, che esprime adeguatamente ed *efficacemente* il grado di somiglianza tra i concetti e le strutture delle due frasi pur rimanendo *indipendente dai linguaggi utilizzati*;

- utilizzando questa metrica, è stato definito e affrontato il problema di *ricerca di similarità tra frasi intere*: individuare, data una frase di query, le frasi di riferimento più simili ad essa, quelle cioé la cui distanza dalla frase cercata non supera una data *soglia*;

- è stato esteso tale problema di ricerca alla più complessa *ricerca di similarità tra parti di frasi*, che si occupa di trovare, utilizzando la medesima metrica di similarità, anche le eventuali *parti* di frasi che più risultano simili a una *qualunque* parte della frase cercata;

- sono stati mappati tali problemi di ricerca in una serie di query, progettate in modo da garantire una buona *efficienza*, grazie all'uso di opportuni *filtri* e *indici*, e un'alta *portabilità*, grazie all'utilizzo di comuni istruzioni SQL e *stored procedure* Java;

- per permettere a tali query di fornire effettivi suggerimenti di traduzione, cioè di suggerire la parte di frase in lingua destinazione corrispondente alla parte di frase in lingua sorgente riconosciuta simile, sono stati studiati ed implementati una serie di algoritmi di *allineamento*, per allineare automaticamente, nel miglior modo possibile e sempre *indipendentemente dalle lingue* usate, prima le frasi all'interno dei paragrafi, poi le parole all'interno delle frasi;

- è stato infine progettato un ambiente comune che riunisce queste funzionalità e ne permette un utilizzo semplice e congiunto, fornendo anche un'interfaccia comune, la possibilità di configurare i vari parametri, nonché la possibilità di gestire ad analizzare una *Translation Memory*.

In questo modo si è ottenuto un ambiente completo, progettato e realizzato in modo da essere, auspicabilmente, sia utile in campo professionale (si pensi ad esempio ai team di traduzione impiegati da Logos), sia di interesse dal punto di vista della ricerca scientifica, proponendo soluzioni inedite e, spesso, riguardanti tematiche poco affrontate.

Per quanto riguarda gli sviluppi futuri, sarebbe interessante:

- studiare estensioni della ricerca di similarità, fino ad ora puramente sintattica, nel campo dell'analisi semantica. Ad esempio, si è valutato l'utilizzo di WordNet [24, 25] per reperire sinonimi delle parole di una frase: si potrebbe estendere l'algortimo di edit distance a considerare uguali due parole che presentano almeno un sinonimo in comune. L'interfacciamento non porrebbe particolari problemi: si è studiata in particolare una interfaccia Java, JWordNet [39], che semplificherebbe notevolmente il problema. Piuttosto, entrare nell'ambito della semantica richiederebbe notevoli sforzi per identificare correttamente la funzione delle parole nella frase e, possibilmente, il loro ambito di utilizzo: considerare simili due parole indipendentemente da tali informazioni risulterebbe nella maggior parte dei casi fuorviante, se non addirittura dannoso. Inoltre, occorrerebbe valutare se i risultati forniti da un utilizzo accorto di strumenti quali WordNet possono essere tanto buoni da giustificare l'abbandono di una caratteristica fondamentale del sistema, l'indipendenza dai linguaggi.

- potenziare gli algoritmi di allineamento, in particolare aggiungendo una sorta di possibilità di apprendimento: l'allineamento delle parole potrebbe ad esempio memorizzare in un database il modo corretto di allineare i termini, in modo da perfezionarne l'efficacia con l'utilizzo, a scapito di una diminuzione di prestazioni dovuta ai tempi di ricerca. Per rendere tale funzione davvero efficace, occorrerebbe effettuare la ricerca sui termini normalizzati, richiedendo pertanto l'utilizzo di moduli di Stemming relativi a ciascuna lingua utilizzata.

- migliorare ulteriormente le prestazioni della ricerca delle sottoparti, lavorando ancora sui filtri proposti e studiandone di nuovi.

- studiare l'applicabilità degli algoritmi di ricerca di similarità e, in generale, di tutto il sistema ad altri ambiti: come si capisce, infatti, la ricerca di similarità è un problema di grande interesse non solo nell'ambito della traduzione multilingua, e gli algoritmi di EXTRA potrebbero, con lievi modifiche, coprire le esigenze di tante altre applicazioni.

Concludendo, il lavoro svolto in questa tesi è stato indubbiamente impegnativo ma anche notevolmente stimolante: è stato possibile approfondire e mettere in pratica le conoscenze acquisite nell'ambito dei Sistemi Informativi e delle Basi di Dati, confrontandosi per la prima volta direttamente con esigenze e problematiche non solo di correttezza, ma anche di efficienza. Sempre per la prima volta si è avuta la possibilità di progettare un software di notevoli dimensioni e di curarne lo sviluppo completo, dalle funzionalità principali all'interfaccia, approfondendo tra l'altro il linguaggio di programmazione Java e l'utilizzo di DBMS commerciali quali Oracle. Lo stesso campo di applicazione, quello dell'information retrieval e della traduzione assistita, è stato un argomento di studio particolarmente insolito ed affascinante: la ricerca in questo contesto è specialmente attiva ed entrare a farne parte proponendo risultati nuovi e, ci auguriamo, di interesse [22] è stato sicuramente un motivo di grande soddisfazione.

Parte III

Appendici

Appendice A

Il codice JAVA (selezione)

Data la notevole estensione del codice Java (oltre 6000 linee), esso viene per completezza comunque riportato, ma solo per le classi più significative, escludendo tra l'altro il codice per la gestione dell'interfaccia grafica.

A.1 Package SimSearch

A.1.1 Classe SimSearch.DBConnect

```
package SimSearch;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                    DBConnect
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
      Classe che contiene le funzioni che si
      interfacciano con il DBMS Oracle
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import oracle.sqlj.runtime.Oracle;
import java.sql.*;
import java.util.*;
import java.text.*;
import java.io.*;
import Stemming.eng.*;

public class DBConnect
 {
   /**
      Costruttore che si connette a Oracle
   */
```

```
DBConnect() throws SQLException
 {
  Oracle.connect(getClass(), "connect.properties");
 }


/**
   Disconnessione da Oracle
*/

public static void disconnect() throws SQLException
 {
   Oracle.close();
 }


/**
   Elimina una tabella delle frasi
   (nome fornito in input)
*/

public static void dropPTable (String tabella) throws SQLException
 {
              // controlla che non venga eliminata la tabella permanente TERMS
   if (tabella.equalsIgnoreCase("TERMS"))
      {
       throw new SQLException("Impossible to drop TERMS table!");
      }

   // Crea una connessione JDBC dal DefaultContext SQLJ corrente
   Connection conn =
   sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

   Statement stmt = conn.createStatement();
   String elimina = "DROP TABLE " + tabella;

   stmt.execute(elimina);          // Elimina la tabella
   stmt.close();
 }


/**
   Elimina le tabelle dei qgrammi associate alla tabella
   delle frasi fornita in input
*/

public static void dropQTables (String tabella) throws SQLException
 {
   String tabella1 = "Q"+tabella;   // il nome delle tabelle dei qgrammi associate
   String tabella2 = "QS"+tabella;

   // Crea una connessione JDBC dal DefaultContext SQLJ corrente
   Connection conn =
```

```
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    stmt.execute ("DROP TABLE " + tabella1);     // Elimina le tabelle
    stmt.execute ("DROP TABLE " + tabella2);

    stmt.close ();
  }


/**
    Crea una tabella delle frasi
    (nome fornito in input)
*/

public static void createPTable (String tabella) throws SQLException
  {
    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String crea = "CREATE TABLE " + tabella
                  +" (codice NUMBER (10) PRIMARY KEY,"
                   +" fraseorig VARCHAR2 (1000),"
                   +" frase  VARCHAR2 (1000),"
                   +" frasetrad VARCHAR2 (1000),"
                   +" wordlen VARCHAR2 (3),"
                   +" apos VARCHAR2 (700),"
                   +" ascore VARCHAR2 (700),"
                   +" spos VARCHAR2 (700)"
                   +")";

    stmt.execute (crea);          // Crea la tabella
    stmt.close ();
  }


/**
    Crea una tabella dei q-grammi
    collegata alla tabella delle frasi fornita in input,
    con larghezza della colonna QGRAM adeguata al
    parametro q (numero di parole nel q-gramma)
*/

public static void createQTable (String tabSorg, int q, boolean qSub)
                                                        throws SQLException
  {
    final int WORDSIZE = Main.MainFrame.WORDSIZE;
                                    // Dimensione delle parole nella frase
                                    // per il calcolo della larghezza max delle
```

```
                                    // colonne dei q−grammi

    int size = q * WORDSIZE; // Dimensione della colonna qgram

    String tabDest = new String();
    if (qSub) tabDest = "QS"+tabSorg;
    else tabDest = "Q"+tabSorg;

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String crea =
          "CREATE TABLE " + tabDest
            +" (codice   NUMBER(10) REFERENCES "+ tabSorg +" ON DELETE CASCADE,"
              +"pos      NUMBER (3),"
              +"qgram    VARCHAR2 (" + size + "),"
              +"PRIMARY KEY (codice,pos)"
              +")";

    String crea2 =
          "CREATE INDEX " + tabDest
            +" ON " + tabDest + " (qgram)";

    stmt.execute (crea);          // Crea la tabella
    stmt.execute (crea2);         // Crea gli indici
    stmt.close ();

  }


/**
   Crea le tabelle dei risultati
*/

public static void createResTables () throws SQLException
  {
    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String crea1 =
          "CREATE TABLE FULLMATCH"
             +"(cod2    NUMBER(10),"
              +"cod1    NUMBER(10),"
              +"dist    NUMBER(3),"
              +"PRIMARY KEY (cod2, cod1),"
              +"FOREIGN KEY (cod1) REFERENCES PHRASE1 (codice) ON DELETE CASCADE,"
              +"FOREIGN KEY (cod2) REFERENCES PHRASE2 (codice) ON DELETE CASCADE"
```

```
            +")";

    String crea2 =
          "CREATE TABLE MATCHPOS"
            +"(cod1    NUMBER(10),"
             +"cod2    NUMBER(10),"
             +"nr      NUMBER(3),"
             +"nc      NUMBER(3),"
             +"PRIMARY KEY (cod1, cod2, nr, nc),"
             +"FOREIGN KEY (cod1) REFERENCES PHRASE1 (codice) ON DELETE CASCADE,"
             +"FOREIGN KEY (cod2) REFERENCES PHRASE2 (codice) ON DELETE CASCADE"
            +")";

    String crea3 =
          "CREATE TABLE SUBMATCH"
            +"(cod2    NUMBER(10),"
             +"i2      NUMBER(3),"
             +"f2      NUMBER(3),"
             +"cod1    NUMBER(10),"
             +"i1      NUMBER(3),"
             +"f1      NUMBER(3),"
             +"dist    NUMBER(3),"
             +"PRIMARY KEY (cod2, i2, cod1, i1),"
             +"FOREIGN KEY (cod1, i1, cod2, i2) REFERENCES "
                    +"MATCHPOS (cod1, nr, cod2, nc) ON DELETE CASCADE,"
             +"FOREIGN KEY (cod1, f1, cod2, f2) REFERENCES "
                    +"MATCHPOS (cod1, nr, cod2, nc) ON DELETE CASCADE,"
             +"FOREIGN KEY (cod2) REFERENCES PHRASE2 (codice) ON DELETE CASCADE"
            +")";

    stmt.execute (crea1 );          // Crea la tabella FULLMATCH
    stmt.execute (crea2 );          // Crea la tabella MATCHPOS
    stmt.execute (crea3 );          // Crea la tabella SUBMATCH

    stmt.close ();
  }


/**
   Elimina le tabelle dei risultati
*/

public static void dropResTables () throws SQLException
  {
    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String elimina1 = "DROP TABLE FULLMATCH";
    String elimina2 = "DROP TABLE SUBMATCH";
    String elimina3 = "DROP TABLE MATCHPOS";
```

```
    stmt.execute (elimina1 );     // Elimina la tabella FULLMATCH
    stmt.execute (elimina2 );     // Elimina la tabella SUBMATCH
    stmt.execute (elimina3 );     // Elimina la tabella MATCHPOS

    stmt.close ();
  }


/**
   Inserisce le frasi presenti nel file , estratte e normalizzate (se stem=TRUE),
   nella tabella delle frasi fornita in input
*/

public static Result fillPTable (Vector vPhrases, String tabella , boolean stem)
                                                              throws SQLException
 {
   java. util .Date data1 = new java.util.Date();       // per calcolo tempo
   long inizio = data1.getTime();

   int phraseCount=0;

   // Crea una connessione JDBC dal DefaultContext SQLJ corrente
   Connection conn =
   sqlj .runtime.ref .DefaultContext.getDefaultContext().getConnection();

   PreparedStatement pstmt = conn.prepareStatement
                    ("INSERT INTO " + tabella + " VALUES (?, ?, ?, ?, ?, ?, ?, ?)");

   for (phraseCount=0; phraseCount<vPhrases.size(); phraseCount++)
       {
         Main.MainFrame.setStatusBar("Extracting and elaborating phrases ("
                        +phraseCount+" of "+(vPhrases.size())+") ...");

         String phraseOrig = (String) vPhrases.elementAt(phraseCount);

                                  // restituisce :
                                  // la frase normalizzata (se stem=true) oppure
                                  // la frase senza punteggiatura (se stem=false)
         Output elabOut = Utility.elabPhrase(phraseOrig, stem);
         String phraseElab = elabOut.getPhrase();

         pstmt.setInt (1, phraseCount+1);
         pstmt.setString (2, phraseOrig);
         pstmt.setString (3, phraseElab);
         pstmt.setString (4, "");
         pstmt.setInt (5, DBUtility.wordLen(phraseElab));
         pstmt.setString (6, "");
         pstmt.setString (7, "");
         pstmt.setString (8, elabOut.getSPos());

         pstmt.execute ();          // Inserisce la frase
       }
```

```
    pstmt.close();

    java.util.Date data2 = new java.util.Date();     // calcolo tempo impiegato
    long fine = data2.getTime();
    long durata = (fine - inizio) / 1000;

    return (new Result (phraseCount, durata));       // numero frasi inserite
                                                     // e tempo impiegato
  }


/**
    Inserisce le frasi della translation memory passate come parametro
    nella tabella delle frasi fornita in input (translation memory)
*/

public static Result memToDB (Vector vTMPhrases, String tabella, int startCod,
                              boolean stem) throws SQLException
  {
    java.util.Date data1 = new java.util.Date();       // per calcolo tempo
    long inizio = data1.getTime();

    int phraseCount=0;

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    PreparedStatement pstmt = conn.prepareStatement
                        ("INSERT INTO " + tabella + " VALUES (?, ?, ?, ?, ?, ?, ?, ?)");

    for (phraseCount=0; phraseCount<vTMPhrases.size(); phraseCount++)
        {
          Main.MainFrame.setStatusBar("Extracting and elaborating phrases ("
                          +phraseCount+" of "+(vTMPhrases.size())+") ...");

          GlobalUtility.TMPhrase tmPhrase = (GlobalUtility.TMPhrase)
                                                vTMPhrases.elementAt(phraseCount);

          String phraseOrig = tmPhrase.getPhraseOrig();
          String phraseTrad = tmPhrase.getPhraseTrad();
          String aPos = tmPhrase.getAPos();
          String aScore = tmPhrase.getAScore();

                                  // restituisce :
                                  // la frase normalizzata (se stem=true) oppure
                                  // la frase senza punteggiatura (se stem=false)
          Output elabOut = Utility.elabPhrase(phraseOrig, stem);
          String phraseElab = elabOut.getPhrase();

          pstmt.setInt (1, startCod);
          pstmt.setString (2, phraseOrig);
```

```
            pstmt.setString (3, phraseElab);
            pstmt.setString (4, phraseTrad);
            pstmt.setInt (5, DBUtility.wordLen(phraseElab));
            pstmt.setString (6, aPos);
            pstmt.setString (7, aScore);
            pstmt.setString (8, elabOut.getSPos());

            pstmt.execute (); // Inserisce la frase

            startCod++;
          }

    pstmt.close ();

    java.util.Date data2 = new java.util.Date(); // calcolo tempo impiegato
    long fine = data2.getTime();
    long durata = (fine - inizio) / 1000;

    return (new Result (phraseCount, durata)); // numero frasi inserite
                                               // e tempo impiegato
  }


/**
   Restituisce il codice maggiore presente nella tabella
   delle frasi fornita in input
*/

public static int getMaxCod (String tabella) throws SQLException
  {
    int maxCod=-1;

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    ResultSet rset = stmt.executeQuery ("SELECT MAX(CODICE) FROM "+tabella);

    if (rset.next())
     {
       maxCod = rset.getInt(1);
     }

    rset.close ();
    stmt.close ();

    return maxCod;
  }


/**
```

```
    Restituisce la distribuzione delle lunghezze delle frasi
    presenti nella tabella passata come parametro
*/

public static int[] getLengthDistrib (String tabella) throws SQLException
 {
   final int MAXLEN = 500;

   int len[] = new int[MAXLEN];

   for (int i=0; i<MAXLEN; i++)
      {
       len[i]=0;
      }

   // Crea una connessione JDBC dal DefaultContext SQLJ corrente
   Connection conn =
   sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

   Statement stmt = conn.createStatement ();
   ResultSet rset = stmt.executeQuery ("SELECT WORDLEN FROM "+tabella);

   while (rset.next())                    // calcola l'array della distribuzione
    {
      int curLen = rset.getInt(1);
      if (curLen >=0 && curLen <MAXLEN)
         len[curLen]++;
    }

   rset.close();
   stmt.close();

   return len;
 }


/**
    Inserisce i q-grammi nella tabella destinazione partendo dalle frasi
    della tabella delle frasi fornita in input
*/

public static Result fillQTable (String tabSorg, int startCod, int q, boolean qSub)
                                                        throws SQLException
 {
   java.util.Date data1 = new java.util.Date();       // per calcolo tempo
   long inizio = data1.getTime();

   String tabDest = new String();
   if (qSub) tabDest = "QS"+tabSorg;
   else tabDest = "Q"+tabSorg;

   String frase = new String();
   Vector vqgram = new Vector();
```

```
int qCount = 0;              // conteggio qgrammi
int pCount = 0;              // conteggio frasi

// Crea una connessione JDBC dal DefaultContext SQLJ corrente
Connection conn =
sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

Statement stmt = conn.createStatement ();
stmt.execute("DROP INDEX " + tabDest);  // elimina l'indice (verr ricreato
                                        // dopo gli inserimenti: pi efficiente )
String query = "SELECT codice, frase"
             +" FROM "+tabSorg
             +" WHERE codice >= "+startCod;

ResultSet rset = stmt.executeQuery (query);

PreparedStatement pstmt = conn.prepareStatement
                ("INSERT INTO " + tabDest + " VALUES (?, ?, ?)");

while (rset.next())
 {
   if (qSub) Main.MainFrame.setStatusBar("Extracting q-grams (sub) (Phrase "
                                    +pCount+", q-gram "+qCount+") ...");
   else Main.MainFrame.setStatusBar("Extracting q-grams (Phrase "
                                    +pCount+", q-gram "+qCount+") ...");

   int cod = rset.getInt("codice");
   frase = rset.getString("frase");

   if (frase != null)     // se c'e' una frase
       {
         vqgram = Utility.qgramVector (frase, q, true);

         for (int i = 0; i < vqgram.size(); i++)
             {PosQgram qg = (PosQgram) vqgram.elementAt (i);
              int pos = qg.getPos();
              String qgram = qg.getQgram();

              pstmt.setInt (1, cod);
              pstmt.setInt (2, pos);
              pstmt.setString (3, qgram);

              pstmt.execute ();
              qCount++;
             }
       }
   pCount++;
 }

Main.MainFrame.setStatusBar("Extracting q-grams (updating indexes)");
stmt.execute("CREATE INDEX " + tabDest
             +" ON " + tabDest + " (qgram)");
```

```
    stmt.close ();
    pstmt.close ();
    rset . close ();

    java. util .Date data2 = new java.util.Date();    // calcolo tempo impiegato
    long fine = data2.getTime();
    long durata = (fine − inizio ) / 1000;

    return (new Result (qCount, durata));     // numero frasi inserite
                                              // e tempo impiegato
  }


/**
   Esegue la query di Approximate String Join
*/

public static Result approxJoinQuery (String tab1, String tab2, int q, double k,
        boolean countFilter, boolean posFilter, boolean lenFilter, boolean editDist,
        boolean subStr) throws SQLException, IllegalArgumentException
 {
                                                    // controllo su K
    if (k<0) throw new IllegalArgumentException
                              ("K must be >= 0");

    java. util .Date data1 = new java.util.Date();      // per calcolo tempo
    long inizio = data1.getTime();

    String qtab1 = "Q"+tab1;
    String qtab2 = "Q"+tab2;

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj .runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String approxQuery =
      "INSERT INTO FULLMATCH "
       +"SELECT r2.codice AS cod2, r1.codice AS cod1,"
                  +" wordEditDistanceDiag (r1.frase, r2.frase , " +k+")"
      +" FROM "+tab1+" r1, "+tab2+" r2";

    if ( posFilter || countFilter ) approxQuery = approxQuery
      +", "+qtab1+" r1q, "+qtab2+" r2q"
      +" WHERE r1.codice = r1q.codice"
      +" AND r2.codice = r2q.codice"
      +" AND r1q.qgram = r2q.qgram";

    if ( posFilter ) approxQuery = approxQuery
     + " AND ABS (r1q.pos − r2q.pos) <= ROUND("+k+"∗ r2.wordLen)";
```

```
    if ( lenFilter )
       {
        if ((! posFilter) && (!countFilter)) approxQuery = approxQuery
     + " WHERE";
        else approxQuery = approxQuery
     + " AND";
        approxQuery = approxQuery
     + " ABS (r1.wordLen - r2.wordLen)"
                        +" <= ROUND("+k+"* r2.wordLen)";
       };

    approxQuery = approxQuery
      +" GROUP BY r2.codice, r1.codice, r1.frase, r2.frase, r1.wordLen, r2.wordLen";

    if (countFilter || editDist) approxQuery = approxQuery
      +" HAVING ";

    if (countFilter) approxQuery = approxQuery
      +" COUNT(*) >= "
            +"(r1.wordLen - 1 - (ROUND("+k+"* r2.wordLen) - 1) * "+q+")"
      +" AND COUNT(*) >= "
            +"(r2.wordLen - 1 - (ROUND("+k+"* r2.wordLen) - 1) * "+q+")";

    if (countFilter && editDist) approxQuery = approxQuery
      +" AND ";

    if (editDist)
       {if (subStr == false)
        approxQuery = approxQuery
      + "wordEditDistanceDiag (r1.frase, r2.frase, "+k+") >= 0";
        else
        approxQuery = approxQuery
      + "wordEditDistanceSubCheck (r1.frase, r1.codice, r2.frase, r2.codice, "+k+") = 0";
        }

    stmt.execute (approxQuery);
    stmt.close ();

    java.util.Date data2 = new java.util.Date();    // calcolo tempo impiegato
    long fine = data2.getTime();
    long durata = (fine - inizio) / 1000;

    return (new Result (0, durata));
  }


/**
   Esegue la query supplementare di Approximate String Join
*/

public static Result approxJoinQueryPass2 (String tab1, String tab2, int q, double k,
                        double kSub, int lMinSub, boolean countFilter, boolean qSub)
                        throws SQLException, IllegalArgumentException
```

```
{
                                                    // controllo su K
  if (k<0 || kSub<0) throw new IllegalArgumentException
                              ("K and KSUB must be >= 0");

  java.util.Date data1 = new java.util.Date();      // per calcolo tempo
  long inizio = data1.getTime();

  String qtab1 = new String();              // le tabelle dei q-grammi da utilizzare
  String qtab2 = new String();
  if (qSub)
     {qtab1 = "QS"+tab1;
      qtab2 = "QS"+tab2;
     }
  else
     {qtab1 = "Q"+tab1;
      qtab2 = "Q"+tab2;
     }

  int minCount = lMinSub - 1 - ((int) Math.round(kSub*lMinSub) - 1) * q - (q - 1) * 2;
  if (minCount < 1) minCount = 1;

  // Crea una connessione JDBC dal DefaultContext SQLJ corrente
  Connection conn =
  sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

  Statement stmt = conn.createStatement ();

  String approxQuery =
    "INSERT INTO FULLMATCH "
     +"SELECT r2.codice AS cod2, r1.codice AS cod1,"
          +" wordEditDistanceDiag (r1.frase, r2.frase, " +k+")"
    +" FROM "+tab1+" r1, "+tab2+" r2";

  if (countFilter) approxQuery = approxQuery
            +", "+qtab1+" r1q, "+qtab2+" r2q";

  approxQuery = approxQuery
    +" WHERE r2.codice NOT IN (SELECT cod2 FROM FULLMATCH)";

  if (countFilter) approxQuery = approxQuery
    +" AND r1.codice = r1q.codice"
    +" AND r2.codice = r2q.codice"
    +" AND r1q.qgram = r2q.qgram";

  approxQuery = approxQuery
    +" GROUP BY r2.codice, r1.codice, r1.frase, r2.frase"
    +" HAVING";

  if (countFilter) approxQuery = approxQuery
    +" COUNT(*) >= "+minCount
    +" AND";
```

```
    approxQuery = approxQuery
      +" wordEditDistanceSubCheck (r1.frase, r1.codice, r2.frase , r2.codice , " +k+") = 0";

    stmt.execute (approxQuery);
    stmt.close ();

    java. util .Date data2 = new java.util.Date();     // calcolo tempo impiegato
    long fine = data2.getTime();
    long durata = (fine - inizio ) / 1000;

    return (new Result (0, durata));
  }


/**
   Trova i match tra le sottostringhe
*/

public static Result findSubMatches (String tab1, String tab2, double kSub, int lMinSub)
                                    throws SQLException, IllegalArgumentException

 {
                                                    // controllo su kSub
    if (kSub<0) throw new IllegalArgumentException
                                ("KSUB must be >= 0");

    java. util .Date data1 = new java.util.Date();       // per calcolo tempo
    long inizio = data1.getTime();

    // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj .runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String subMatchQuery =
      "INSERT INTO SUBMATCH"
     +" SELECT m1.cod2, m1.nc, m2.nc, m1.cod1, m1.nr, m2.nr,"
               +" wordEditDistanceDiag (wordSubString(p1.frase, m1.nr, m2.nr),"
               +" wordSubString(p2.frase, m1.nc, m2.nc), "+kSub+")"
     +" FROM MATCHPOS m1, MATCHPOS m2, "+tab1+" p1, "+tab2+" p2"
     +" WHERE m1.cod1 = m2.cod1"
     +" AND m1.cod2 NOT IN (SELECT cod2 FROM FULLMATCH)"
     +" AND m1.cod2 = m2.cod2"
     +" AND m1.cod1 = p1.codice"
     +" AND m1.cod2 = p2.codice"
     +" AND m1.nr < m2.nr"
     +" AND m1.nc < m2.nc"
     +" AND (m2.nc - m1.nc + 1) >= "+lMinSub
     +" AND (m2.nr - m1.nr + 1) >= "+lMinSub
     +" AND ABS((m2.nc - m1.nc) - (m2.nr - m1.nr)) <= ROUND((m2.nc - m1.nc +1)*"+kSub+")"
     +" AND (SELECT COUNT(*)"
         +" FROM MATCHPOS m1a"
```

```
    +" WHERE m1a.cod1 = m1.cod1"
    +" AND m1a.cod2 = m1.cod2"
    +" AND m1a.nc <= m2.nc"
    +" AND m1a.nc >= m1.nc"
    +" AND m1a.nr <= m2.nr"
    +" AND m1a.nr >= m1.nr"
    +" AND ABS((m1a.nc - m1.nc)-(m1a.nr - m1.nr))"
          +" <=ROUND((m2.nc - m1.nc +1)*"+kSub+")"
     +") >= ALL ((m2.nc - m1.nc +1) - ROUND((m2.nc - m1.nc +1)*"+kSub+"),"
           +"(m2.nr - m1.nr +1) - ROUND((m2.nc - m1.nc +1)*"+kSub+"))"
+" AND wordEditDistanceDiag (wordSubString(p1.frase, m1.nr, m2.nr),"
         +" wordSubString(p2.frase, m1.nc, m2.nc), "+kSub+") >= 0"
+" AND NOT EXISTS "
    +" (SELECT m3.nc, m3.nr, m4.nc, m4.nr"
     +" FROM MATCHPOS m3, MATCHPOS m4, "+tab1+" p3, "+tab2+" p4"
     +" WHERE m3.cod1 = m4.cod1"
     +" AND m3.cod2 = m4.cod2"
     +" AND m3.cod1 = p3.codice"
     +" AND m3.cod2 = p4.codice"
     +" AND m3.nr < m4.nr"
     +" AND m3.nc < m4.nc"
     +" AND (m4.nr - m3.nr + 1) >= "+lMinSub
     +" AND ABS((m4.nc - m3.nc)-(m4.nr - m3.nr))"
          +" <=ROUND((m4.nc - m3.nc +1)*"+kSub+")"
     +" AND (SELECT COUNT(*)"
       +" FROM MATCHPOS m3a"
       +" WHERE m3a.cod1 = m3.cod1"
       +" AND m3a.cod2 = m3.cod2"
       +" AND m3a.nc <= m4.nc"
       +" AND m3a.nc >= m3.nc"
       +" AND m3a.nr <= m4.nr"
       +" AND m3a.nr >= m3.nr"
       +" AND ABS((m3a.nc - m3.nc)-(m3a.nr - m3.nr))"
            +" <=ROUND("+kSub+"*(m4.nc-m3.nc +1))"
        +") >= ALL ((m4.nc - m3.nc +1) - ROUND((m4.nc - m3.nc +1)*"+kSub+"),"
                +"(m4.nr - m3.nr +1) - ROUND((m4.nc - m3.nc +1)*"+kSub+"))"
     +" AND wordEditDistanceDiag (wordSubString(p3.frase, m3.nr, m4.nr),"
              +" wordSubString(p4.frase, m3.nc, m4.nc), "+kSub+") >= 0"
     +" AND m3.cod2 = m1.cod2"
     +" AND ((m1.nc = m3.nc AND m4.nc > m2.nc)"
          +" OR"
          +" (m3.nc < m1.nc AND m2.nc = m4.nc)"
          +" OR"
          +" (m3.nc < m1.nc AND m4.nc > m2.nc)"
          +" OR"
          +" (m1.nc = m3.nc AND m2.nc = m4.nc AND m1.cod1 = m3.cod1 AND"
            +" ((m1.nr = m3.nr AND m4.nr > m2.nr)"
              +" OR"
             +" (m3.nr < m1.nr AND m2.nr = m4.nr)"
              +" OR"
             +" (m3.nr < m1.nr AND m4.nr > m2.nr)"
             +")"
         +")"
```

```
                +")"
            +")";

    stmt.execute (subMatchQuery);

    stmt.close ();

    java.util.Date data2 = new java.util.Date();    // calcolo tempo impiegato
    long fine = data2.getTime();
    long durata = (fine - inizio) / 1000;

    return (new Result (0, durata));
  }


/**
   Estrae i risultati
*/

public static Result extractResults (String tab1, String tab2, FileWriter outputFile,
                    boolean subStr, Vector vPhrases, String dDelim, String sDelim)
                    throws SQLException, IOException

 {
    java.util.Date data1 = new java.util.Date();      // per calcolo tempo
    long inizio = data1.getTime();

    PrintWriter pwOutFile = new PrintWriter (outputFile);

        // Crea una connessione JDBC dal DefaultContext SQLJ corrente
    Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

    Statement stmt = conn.createStatement ();

    String resultQuery =
          "SELECT fm.cod2 AS codice2,"
              +" p2.frase AS frase2 , "  AS sfrase2,"
              +" p2.fraseorig AS fraseorig2 , "  AS sfraseorig2,"
              +" fm.cod1 AS codice1,"
              +" p1.frase AS frase1 , "  AS sfrase1,"
              +" p1.fraseorig AS fraseorig1 , "  AS sfraseorig1,"
              +" p1.frasetrad AS frasetrad1 , "  AS sfrasetrad1,"
              +" fm.dist AS dist, (fm.dist / p2.wordLen) AS distrel,"
              +" 0 AS inizio, 0 AS sub"
        +" FROM FULLMATCH fm, "+tab1+" p1, "+tab2+" p2"
        +" WHERE fm.cod1 = p1.codice"
        +" AND fm.cod2 = p2.codice";

    if (subStr) resultQuery = resultQuery              // parte per sottostringhe
        +" UNION"
        +" SELECT sm.cod2 AS codice2,"
              +" p2.frase AS frase2,"
```

```
            +" wordSubString(p2.frase, sm.i2, sm.f2) AS sfrase2,"
            +" p2.fraseorig AS fraseorig2,"
            +" wordSubString(p2.fraseorig, transPos(p2.spos,sm.i2), transPos(p2.spos,sm.f2))"
                    +" AS sfraseorig2,"
            +" sm.cod1 AS codice1,"
            +" p1.frase AS frase1,"
            +" wordSubString(p1.frase, sm.i1, sm.f1) AS sfrase1,"
            +" p1.fraseorig AS fraseorig1,"
            +" wordSubString(p1.fraseorig, transPos(p1.spos,sm.i1), transPos(p1.spos,sm.f1))"
                    +" AS sfraseorig1,"
            +" p1.frasetrad AS frasetrad1,"
            +" wordSubString(p1.frasetrad, transPos(p1.apos,transPos(p1.spos,sm.i1)),"
                    +"transPos(p1.apos,transPos(p1.spos,sm.f1))) AS sfrasetrad1,"
            +" sm.dist AS dist, (sm.dist / (sm.f2-sm.i2+1)) AS distrel,"
            +" sm.i2 AS inizio, 1 AS sub"
        +" FROM SUBMATCH sm, "+tab1+" p1, "+tab2+" p2"
        +" WHERE sm.cod1 = p1.codice"
        +" AND sm.cod2 = p2.codice";

resultQuery = resultQuery
        +" ORDER BY codice2, sub, inizio, dist, codice1";

ResultSet rset = stmt.executeQuery (resultQuery);

int cercata = 0;                    // codice ultima frase cercata

int fullMatchCount = 0;
int subMatchCount = 0;
int phrasesTot = vPhrases.size();
int phrasesFull = 0;
int phrasesSub = 0;

while (rset.next())                 // estrazione risultati
   {
     int cur = rset.getInt("codice2");      // frase cercata corrente

     for (int i=cercata+1; i<cur; i++)
        {
         if (i != 1)
            {
             pwOutFile.println("##\r\n<Insert translation here>");
             pwOutFile.println(dDelim+" 0");
             pwOutFile.println(sDelim);
            }
         pwOutFile.println(
          "##\r\n## ----------------------------------------");
         pwOutFile.println("## PHRASE NOT FOUND: (COD " + i
                        + ") "+ ((String) vPhrases.elementAt(i-1)));
        }

     if (cercata != cur)      // stampa la frase cercata
                              // se diversa dalla precedente
        {
```

```
        cercata = cur;

        if (cercata != 1)
           {
            pwOutFile.println("##\r\n<Insert translation here>");
            pwOutFile.println(dDelim+" 0");
            pwOutFile.println(sDelim);
           }

        pwOutFile.println(
         "##\r\n## ----------------------------------------");
        pwOutFile.println("## PHRASE FOUND: (COD " + cercata
                    + ") "+ rset.getString("frase2"));
        pwOutFile.println("##                    (ORIG) " + rset.getString("fraseorig2"));

        if (rset.getInt("sub") == 0) phrasesFull++; // conteggio frasi trovate full
        else phrasesSub++;                          // conteggio frasi trovate sub
       };

    if (rset.getInt("sub") == 0)                    // full match
       {
        pwOutFile.println("## FULL(D="+rset.getInt("dist")+"/RD="
                    +(new DecimalFormat("0.00").format(rset.getDouble("distrel")))
                    +"):(COD "+rset.getInt("codice1")+ ") "+rset.getString("frase1"));
        pwOutFile.println("##                    (ORIG) " + rset.getString("fraseorig1"));
        pwOutFile.println("##                    (TRAD -->) "
                                                + rset.getString("frasetrad1"));

        fullMatchCount ++;
       }
    else                                            // sub match
       {
        pwOutFile.println("## SUB (D="+rset.getInt("dist")+"/RD="
                    +(new DecimalFormat("0.00").format(rset.getDouble("distrel")))
                    +"):(I="+rset.getString("inizio")+") "+rset.getString("sfrase2"));
        pwOutFile.println("##                    (ORIG) "+rset.getString("sfraseorig2"));
        pwOutFile.println("##                    (COD "+rset.getInt("codice1")
                    + ") "+rset.getString("sfrase1"));
        pwOutFile.println("##                    (ORIG) "+rset.getString("sfraseorig1"));
        pwOutFile.println("##                    (TRAD -->) "
                                                 +rset.getString("sfrasetrad1"));
        pwOutFile.println("##                    (FULL) " + rset.getString("frase1"));
        pwOutFile.println("##                    (FULL ORIG) "
                                                 +rset.getString("fraseorig1"));
        pwOutFile.println("##                    (FULL TRAD) "
                                                 +rset.getString("frasetrad1"));

        subMatchCount ++;
       }
   }

pwOutFile.println("##\r\n<Insert translation here>");
pwOutFile.println(dDelim+" 0");
```

```
        pwOutFile.println(sDelim);
        pwOutFile.println(
          "##\r\n## ----------------------------------------");

        for (int i=cercata+1; i<=vPhrases.size(); i++)
           {
            pwOutFile.println(
              "##\r\n## ----------------------------------------");
            pwOutFile.println("## PHRASE NOT FOUND: (COD " + i
                              + ") "+ ((String) vPhrases.elementAt(i-1)));
            pwOutFile.println("##\r\n<Insert translation here>");
            pwOutFile.println(dDelim+" 0");
            pwOutFile.println(sDelim);
           }

        rset.close();
        stmt.close();

        java.util.Date data2 = new java.util.Date();    // calcolo tempo impiegato
        long fine = data2.getTime();
        long durata = (fine - inizio) / 1000;

        int phraseCount = fullMatchCount + subMatchCount; // suggerimenti trovati in tutto
        Vector vCount = new Vector();                     // vettore per conteggi dettagliati
        vCount.addElement (new Integer(fullMatchCount));
        vCount.addElement (new Integer(subMatchCount));
        vCount.addElement (new Integer(phrasesTot));
        vCount.addElement (new Integer(phrasesFull));
        vCount.addElement (new Integer(phrasesSub));

        return (new Result (phraseCount, durata, vCount));
      }
  }
```

A.1.2 Classe SimSearch.DBUtility

```
package SimSearch;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                          DBUtility
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
     Classe astratta contenente quella parte di funzioni
     di utilita ' richiamate anche da query del DBMS
     (codice da caricare nel server di Oracle)
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;

public abstract class DBUtility
```

```
{
  /**
     Restituisce  il  minore di tre  valori
  */

  public static int minimum (int a, int b, int c)
   {
     int min;

     min = a;
     if (b < min) { min = b; }
     if (c < min) { min = c; }

     return min;
   }


  /**
    Restituisce un vettore con le parole  estratte
    dalla stringa s in ingresso
  */

  public static Vector vectorFromString (String s)
  {
    StringTokenizer stok = new StringTokenizer(s);
    Vector v = new Vector();

    while (stok.hasMoreTokens())
     {
       v.addElement (stok.nextToken());
     }
    return v;
  }


  /**
    Restituisce  il  numero di parole (lunghezza in parole)
    della  stringa  s
  */

  public static int wordLen (String s)
  {
    if (s == null) return 0;
    if (s.equals("")) return 0;

    StringTokenizer st = new StringTokenizer(s);
    return st.countTokens ();       // numero parole di s
  }


  /**
    Restituisce  la  sottostringa  di  s  che comincia e finisce
    dalle  parole  di  posizione  specificata
```

```
*/

public static String wordSubString (String s, int fromWord, int toWord)
{
  if (s == null) return "";
  if (s.equals("")) return "";

  int startIndex = 0;
  int endIndex = 0;

  if (fromWord > toWord)
     {
      int w = toWord;
      toWord = fromWord;
      fromWord = w;
     }

  for (int i=1; i<fromWord; i++)
      {
       startIndex = s.indexOf(" ", startIndex+1);
      }

  for (int i=0; i<toWord; i++)
      {
       endIndex = s.indexOf(" ", endIndex+1);
      }

  if (startIndex > 0) startIndex++;
  else if (startIndex == -1) startIndex = s.length();

  if (endIndex == -1) endIndex = s.length();

  return s.substring(startIndex, endIndex);      // la sottostringa di s cercata
}


/**
  Restituisce la posizione (numero) della parola della frase originale
  corrispondente alla n-esima parola della frase normalizzata
*/

public static int transPos (String sPos, int n)
{
  if (sPos==null) return n;              // per il caso STEM=false
  if (sPos.equals("")) return n;

  int startIndex = 0;
  int endIndex = 0;

  for (int i=1; i<=n; i++)
      {
       startIndex = sPos.indexOf("<", startIndex)+1;
      }
```

```
    endIndex = sPos.indexOf(">", startIndex);

                                        // la sottostringa di s cercata
    return Integer.parseInt(sPos.substring(startIndex, endIndex));
  }
}
```

A.1.3 Classe SimSearch.Distance

```
package SimSearch;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                        Distance
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
     Classe astratta contenente varie funzioni riguardanti
     il calcolo della edit-distance tra stringhe
     (codice da caricare nel server di Oracle)
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;
import java.sql.*;

public abstract class Distance
{
  /**
    Calcola la Edit Distance (distanza in parole)
    tra due stringhe
  */

  public static int wordEditDistance (String s, String t)
   {
    int n = DBUtility.wordLen (s);     // numero parole di s
    int m = DBUtility.wordLen (t);     // numero parole di t

                                // casi particolari
    if (n == 0) { return m; }
    if (m == 0) { return n; }

    int d [][] = new int[n+1][m+1]; // matrice di costi
    int i;      // itera in s
    int j;      // itera in t
    String s_i;  // iesima parola di s
    String t_j;  // jesima parola di t
    int cost;  // costo

    Vector sv = DBUtility.vectorFromString (s);
    Vector tv = DBUtility.vectorFromString (t);
```

```
    for (i = 0; i <= n; i++)       // inizializzazione prima colonna
          {d[i][0] = i;}
    for (j = 0; j <= m; j++)       // inizializzazione prima riga
          {d[0][j] = j; }

    for (i = 1; i <= n; i++)         // calcolo matrice dei costi
          {s_i = (String) sv.elementAt (i - 1);
           for (j = 1; j <= m; j++)
                {t_j = (String) tv.elementAt (j-1);
                 if ( s_i.compareTo(t_j)==0) { cost = 0; }
                 else { cost = 1; }
                 d[i][j] = DBUtility.minimum
                          (d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1] + cost);
                }
          }

    return d[n][m];
  }


/**
   Calcola la Edit Distance (distanza in parole)
   tra due stringhe
   (versione ottimizzata per calcolo su diagonali)
*/

public static int wordEditDistanceDiag (String s, String t, double maxDist)
  {
    int n = DBUtility.wordLen (s);     // numero parole di s
    int m = DBUtility.wordLen (t);     // numero parole di t

    final int VUOTO = -10;       // valore di inizializzazione array

                                 // casi particolari
    if (n == 0 && m == 0) { return -1; }
    if (n == 0) { return m; }
    if (m == 0) { return n; }

    int [][] d = new int[n+1][m+1]; // matrice di costi

                                    // array dei punti di partenza degli stroke
    int startx[] = new int [m+1];
    int starty[] = new int [n+1];

    int stroke=0;                   // valore dello stroke di partenza

    int i;                          // variabili per cicli
    int j;
    int k;

    int posx = 0;                   // variabili per estremi di calcolo
    int posy = 1;
    int stopx = m;
```

```
int stopy = n;

Vector sv = DBUtility.vectorFromString (s);
Vector tv = DBUtility.vectorFromString (t);

                                    // inizializzazione  array
for (i = 0; i <= n; i++)
    {for (j = 0; j <= m; j++)
        {d[i][j] = VUOTO;}
    }

for (i = 0; i <= n; i++)
      {starty[i] = 0;}
for (j = 0; j <= m; j++)
      {startx[j] = 0;}

                                    // calcolo  della  matrice per stroke  diagonali
for (stroke=0; (stroke<=Math.round(maxDist*m)) ; stroke++)
    {
                          // calcolo  stroke  diagonali  della  matrice triangolare  superiore
     k = posx;
     while (k<=stroke && k<=stopx)
        {
         i = startx[k];
         j = k + startx[k];

         do
             {if (i==n && j==m) return stroke;
              d[i][j] = stroke;

              i++;
              j++;
              if (i>n)
                 {posx=k+1;
                  stopy=-1;
                  break;
                 }
              if (j>m)
                 {stopx=k-1;
                  break;
                 }
             }
         while (((String) sv.elementAt(i-1)).compareTo((String) tv.elementAt(j-1)) == 0
                 || ( d[i-1][j] == stroke-1) || (d[i][j-1] == stroke-1));

         startx[k] = i;
         k++;
        }
                          // calcolo  stroke  diagonali  della  matrice triangolare  inferiore
     k = posy;
     while (k<=stroke && k<=stopy)
        {
         i = k + starty[k];
```

```
            j = starty[k];

            do
                {if (i==n && j==m) return stroke;
                 d[i][j] = stroke;

                 i++;
                 j++;
                 if (i>n)
                    {stopy=k-1;
                     break;
                    }
                 if (j>m)
                    {posy=k+1;
                     stopx=-1;
                     break;
                    }
                }
            while (((String) sv.elementAt(i-1)).compareTo((String) tv.elementAt(j-1)) == 0
                    || ( d[i-1][j] == stroke-1) || (d[i][j-1] == stroke-1));

            starty[k] = j;
            k++;
           }

       }

   return (-1);
 }


/**
   Calcola la Edit Distance (distanza in parole)
   tra due stringhe
   (versione con estrazione risultati parziali per successiva
    ricerca sottoparti simili)
*/

public static int wordEditDistanceSubCheck (String s, int codS, String t, int codT,
                                    double maxDist) throws SQLException
 {
   int n = DBUtility.wordLen (s);     // numero parole di s
   int m = DBUtility.wordLen (t);     // numero parole di t

                                 // casi particolari
   if (n == 0) { return -1; }
   if (m == 0) { return -1; }

   int d [][] = new int[n+1][m+1]; // matrice di costi
   int i;      // itera in s
   int j;      // itera in t
   String s_i;  // iesima parola di s
   String t_j;  // jesima parola di t
```

```
Vector sv = DBUtility.vectorFromString (s);
Vector tv = DBUtility.vectorFromString (t);

Vector matchv = new Vector ();

for (i = 0; i <= n; i++)       // inizializzazione prima colonna
      {d[i][0] = i;}
for (j = 0; j <= m; j++)       // inizializzazione prima riga
      {d[0][j] = j; }

for (i = 1; i <= n; i++)          // calcolo matrice dei costi
      {s_i = (String) sv.elementAt (i - 1);
       for (j = 1; j <= m; j++)
            {t_j = (String) tv.elementAt (j-1);
                                  // parole uguali
             if (s_i.compareTo(t_j)==0)
                {
                 d[i][j]=d[i-1][j-1];

                 int match[] = {i, j};
                 matchv.addElement (match);
                }
             else                 // parole diverse
                {
                 d[i][j] = 1 + DBUtility.minimum (d[i-1][j], d[i][j-1], d[i-1][j-1]);
                }
            }
      }

if (d[n][m] <= Math.round(maxDist*m)) // trovato match tra stringhe (frasi) intere
   {
    return 0;
   }
else
   {
    if (matchv.size()>=2)
                      // inserimento nella tabella MATCHPOS dei dati necessari
                      // per il match tra sottostringhe
      {
       // Crea una connessione JDBC dal DefaultContext SQLJ corrente
       Connection conn =
       sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();

       PreparedStatement pstmt = conn.prepareStatement
                      ("INSERT INTO MATCHPOS VALUES (?, ?, ?, ?)");

       for (i = 0; i < matchv.size(); i++)
         {
          int match[] = (int[]) matchv.elementAt(i);

          pstmt.setInt (1, codS);
          pstmt.setInt (2, codT);
```

```
                pstmt.setInt  (3,  match[0]);
                pstmt.setInt  (4,  match[1]);

                pstmt.execute  ();
               }
              pstmt.close ();
             }
           return −1;
          }
    }
}
```

A.1.4 Classe SimSearch.PosQGram

```
package SimSearch;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                              PosQgram
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
     Classe che implementa l'oggetto Q−gramma posizionale
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

public class PosQgram
{
  private int pos;
  private String qgram;


  /**
    Costruttore
  */

  PosQgram (int p, String s)
   {
    pos = p;
    qgram = s;
   }

  /**
     Restituisce  la  posizione  del  q−gramma
  */

  public int getPos ()
   {
     return pos;
   }


  /**
```

```
    Restituisce la stringa del q-gramma
  */

  public String getQgram ()
   {
     return qgram;
   }


  /**
    Modifica la posizione del q-gramma
  */

  public void setPos (int p)
   {
     pos = p;
   }


  /**
    Modifica la stringa del q-gramma
  */

  public void setQgram (String s)
   {
     qgram = s;
   }
}
```

A.1.5 Classe SimSearch.PretranslationStat

```
package SimSearch;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                    PretranslationStat
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
    Classe che implementa l'oggetto contenente tutti
    i dati statistici sulla pretraduzione effettuata
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;

public class PretranslationStat
{
  private int fullTime;
  private int fullTime2;
  private int subTime;
  private int totTime;
```

```
private int totTMPhrases;
private int totPhrases;
private int fullPhrases;
private int subPhrases;
private int notFoundPhrases;
private int coverage;

private int fullMatches;
private int subMatches;

/**
   Costruttore
*/

PretranslationStat (int ft, int ft2, int st, int ttmp, int tp, int fp, int sp,
                    int fm, int sm)
 {
  fullTime = ft;
  fullTime2 = ft2;
  subTime = st;
  totTime = ft + ft2 + st;

  totTMPhrases = ttmp;
  totPhrases = tp;
  fullPhrases = fp;
  subPhrases = sp;
  notFoundPhrases = tp - fp - sp;
  coverage = (int) (((sp+fp)*100)/tp);

  fullMatches = fm;
  subMatches = sm;
 }

/**
   Restituisce il tempo totale
*/

public int getTotTime()
 { return totTime; }

/**
   Restituisce il tempo per la ricerca full
*/

public int getFullTime()
 { return fullTime; }

/**
   Restituisce il tempo per la ricerca full (seconda passata)
*/

public int getFullTime2()
 { return fullTime2; }
```

```
/**
   Restituisce il tempo per la ricerca sub
*/

public int getSubTime()
 { return subTime; }

/**
   Restituisce il numero di frasi nella Tr Memory
*/

public int getTotTMPhrases()
 { return totTMPhrases; }

/**
   Restituisce il numero di frase pretradotte
*/

public int getTotPhrases()
 { return totPhrases; }

/**
   Restituisce il numero di frasi con full match
*/

public int getFullPhrases()
 { return fullPhrases; }

/**
   Restituisce il numero di frasi con sub match
*/

public int getSubPhrases()
 { return subPhrases; }

/**
   Restituisce il numero di frasi non trovate
*/

public int getNotFoundPhrases()
 { return notFoundPhrases; }

/**
   Restituisce la copertura percentuale
*/

public int getCoverage()
 { return coverage; }

/**
   Restituisce il numero di full match
*/
```

```
  public int getFullMatches()
   { return fullMatches; }

  /**
     Restituisce  il  numero di sub match
  */

  public int getSubMatches()
   { return subMatches; }
}
```

A.1.6 Classe SimSearch.Result

```
package SimSearch;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                              Result
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
     Classe che implementa l'oggetto  risultato  (di una
     elaborazione ),  composto da:
     * un intero (numero operazioni eseguite),
     * un long (tempo impiegato)
     * un eventuale vettore  di  oggetti  ( il   risultato
       vero  e  proprio  dell 'elaborazione  o maggiori
       dettagli  su  di  essa)
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;

public class Result
{
  private int count;
  private long compTime;
  private Vector vResults;


  /**
    Costruttori
  */

  Result (int c, long t)
   {
    count = c;
    compTime = t;
    vResults = null;
   }

  Result (int c, long t, Vector v)
```

```
   {
    count = c;
    compTime = t;
    vResults = v;
   }


  /**
      Restituisce  il  numero di operazioni eseguite
  */

  public int getCount ()
   {
     return count;
   }


  /**
      Restituisce  il  tempo impiegato
  */

  public long getCompTime ()
   {
     return compTime;
   }


  /**
      Restituisce  il  vettore  dei  risultati
  */

  public Vector getResultVector ()
   {
     return vResults;
   }
}
```

A.1.7 Classe SimSearch.SimSearch

```
package SimSearch;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                          SimSearch
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
      Classe che  realizza  la  ricerca  di  somiglianza
      e  gestisce  la  memoria di traduzione
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.sql.*;
```

```
import java.util.*;
import java.io.*;

public class SimSearch
 {
   /**
     Metodo principale per la ricerca di somiglianza
     (pretraduzione) di un file
   */

   public static PretranslationStat pretranslate (boolean oneLine)
                       throws SQLException, IOException, IllegalArgumentException
    {
         // parametri di configurazione ( modificabili nel modulo principale)

      final String FILEIN = Main.MainFrame.FILEIN1; // file di input

      final String TABLE1 = Main.MainFrame.TABLE1; // tabella delle frasi di riferimento
      final String TABLE2 = Main.MainFrame.TABLE2; // tabella delle frasi di query

      final int Q = Main.MainFrame.Q;      // numero parole in un qgramma
      final int QSUB = Main.MainFrame.QSUB;
      final double K = Main.MainFrame.K; // massima distanza consentita (frasi intere)

      final int LMINSUB = Main.MainFrame.LMINSUB; // lunghezza minima sottostringhe
                                          // massima distanza consentita tra sottostringhe
      final double KSUB = Main.MainFrame.KSUB;
                                          // attiva la normalizzazione
      final boolean STEM = Main.MainFrame.STEM;
                                          // attiva il count filter nella query
      final boolean COUNTFILTER = Main.MainFrame.COUNTFILTER;
                                          // attiva il position filter nella query
      final boolean COUNTFILTER2 = Main.MainFrame.COUNTFILTER2;
                                          // attiva il position filter nella query successiva
      final boolean POSFILTER = Main.MainFrame.POSFILTER;
                                          // attiva il length filter nella query
      final boolean LENGTHFILTER = Main.MainFrame.LENGTHFILTER;
                                          // attiva il controllo sulla e.d. nella query
      final boolean EDITDISTANCE = Main.MainFrame.EDITDISTANCE;
                                          // modo di ricerca delle sottostringhe
      final boolean SUBSTR = Main.MainFrame.SUBSTR;
      final int SUBSTRMODE = Main.MainFrame.SUBSTRMODE;
                                          // delimitatori frasi
      final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER;
      final String DISTDELIMITER = Main.MainFrame.DISTDELIMITER;

      FileReader inputFile = null;

      try                                  // apertura del file di input
        {
         inputFile = new FileReader (FILEIN);
        }
      catch (FileNotFoundException e)
```

```
  {
   throw new IOException ("Error opening input file: "+e.getMessage());
  }

Vector vPhrases = new Vector();

Main.MainFrame.setStatusBar("Extracting and elaborating phrases...");

try
 {                                    // lettura del file di input
  if (!oneLine) vPhrases = GlobalUtility.GlobalUtility.readPhrases (inputFile);
  else vPhrases = GlobalUtility.GlobalUtility.readLines (inputFile);
 }
catch (IOException e)
 {throw new IOException ("Error reading file: "+e.getMessage());
 }
finally
 {
  try {inputFile.close ();}           // chiusura del file di input
  catch (IOException e) {}
 }

String fileout1 = FILEIN + ".al";
String fileout2 = FILEIN + "-pretrad"+".al";
FileWriter outputFile1 = null;
FileWriter outputFile2 = null;

try                                  // apertura dei file di output
  {
   outputFile1 = new FileWriter (fileout1);
   outputFile2 = new FileWriter (fileout2);
  }
catch (IOException e)
  {
   throw new IOException
        ("Error creating output file : " +e.getMessage());
  }
                                                 // separazione punteggiatura
vPhrases = GlobalUtility.GlobalUtility.separatePunct (vPhrases);
                                                 // eliminazione doppi spazi
vPhrases = GlobalUtility.GlobalUtility.cleanSpaces (vPhrases);

writePhrases (vPhrases, outputFile1);        // scrittura file di output 1

              // stampa intestazione file di output 2 (di pretraduzione)
PrintWriter pwOutFile2 = new PrintWriter (outputFile2);
pwOutFile2.println("## ------------------------------------------");
pwOutFile2.println("## Q = "+Q);
if (SUBSTR)
  pwOutFile2.println("## QSUB = "+QSUB);
pwOutFile2.println("## K = "+K);
pwOutFile2.println("## ------------------------------------------");
if (SUBSTR)
```

```
 {pwOutFile2.println("## LMINSUB = "+LMINSUB);
  pwOutFile2.println("## KSUB = "+KSUB);
  pwOutFile2.println("## ----------------------------------------");
 };
pwOutFile2.println("## FILEIN = "+FILEIN);
pwOutFile2.println("## ----------------------------------------");
pwOutFile2.println("## STEM = "+STEM);
pwOutFile2.println("## SUBSTR = "+SUBSTR);
if (SUBSTR)
  pwOutFile2.println("## SUBSTRMODE = "+SUBSTRMODE);
pwOutFile2.println("## ----------------------------------------");
pwOutFile2.println("## COUNTFILTER = "+COUNTFILTER);
pwOutFile2.println("## POSFILTER = "+POSFILTER);
pwOutFile2.println("## LENGTHFILTER = "+LENGTHFILTER);
pwOutFile2.println("## EDITDISTANCE = "+EDITDISTANCE);
if (SUBSTR)
  pwOutFile2.println("## COUNTFILTER2 = "+COUNTFILTER2);
pwOutFile2.println("## ----------------------------------------");

DBConnect db = null;
Result ris = null;

try                                          // operazioni sul DB
 {
  db = new DBConnect();
                                             // pulizia tabelle temporanee
  Main.MainFrame.setStatusBar("Cleaning temp tables...");
  db.dropResTables();
  db.dropQTables(TABLE2);
  db.dropPTable(TABLE2);
  db.createPTable(TABLE2);
  db.createQTable(TABLE2, Q, false);
  db.createQTable(TABLE2, QSUB, true);
  db.createResTables();

  ris = db.fillPTable(vPhrases, TABLE2, STEM);
  pwOutFile2.println("##\r\n## Input file: "+FILEIN);
  pwOutFile2.println("## Phrases extracted: "+ris.getCount());
  pwOutFile2.println("## Seconds used: "+ris.getCompTime());
  if (STEM) pwOutFile2.println("## (with stemming)");
  else pwOutFile2.println("## (without stemming)");

  Main.MainFrame.setStatusBar("Extracting q-grams...");

  ris = db.fillQTable(TABLE2, 1, Q, false);
  pwOutFile2.println("##\r\n## Q-grams extracted: "+ris.getCount());
  pwOutFile2.println("## Seconds used: "+ris.getCompTime()+"\r\n##");

  if (QSUB != Q)
     {ris = db.fillQTable(TABLE2, 1, QSUB, true);
      pwOutFile2.println("## Q-grams extracted (sub): "+ris.getCount());
      pwOutFile2.println("## Seconds used: "+ris.getCompTime()+"\r\n##");
     }
```

```
int fullTime = 0;          // tempo impiegato nella ricerca dei full match
int fullTime2 = 0;         // tempo impiegato nella ricerca dei full match
                           // (seconda passata)

Main.MainFrame.setStatusBar("Executing approxStringJoin query...");
if ((!SUBSTR) || (SUBSTR && SUBSTRMODE==1)) // no substring o substring 1-pass
   {ris = db.approxJoinQuery(TABLE1, TABLE2, Q, K, COUNTFILTER, POSFILTER,
                                          LENGTHFILTER, EDITDISTANCE, SUBSTR);
    fullTime = (int) ris.getCompTime();
    pwOutFile2.println("## Seconds used (ApproxStringJoin): "+fullTime);
   }
else if ((SUBSTRMODE==2) || (QSUB==Q)) // substring 2-pass (fast)
   {ris = db.approxJoinQuery(TABLE1, TABLE2, Q, K, COUNTFILTER, POSFILTER,
                                          LENGTHFILTER, EDITDISTANCE, false);
    fullTime = (int) ris.getCompTime();
    pwOutFile2.println("## Seconds used (ApproxStringJoin): "+fullTime);

    Main.MainFrame.setStatusBar("Executing approxStringJoin Pass2 query ...");
    ris = db.approxJoinQueryPass2(TABLE1, TABLE2, Q, K, KSUB, LMINSUB,
                                          COUNTFILTER2, false);
    fullTime2 = (int) ris.getCompTime();
    pwOutFile2.println("## Seconds used (ApproxSJ Pass2): "+fullTime2);
   }
else                                          // substring 2-pass (accurate)
   {ris = db.approxJoinQuery(TABLE1, TABLE2, Q, K, COUNTFILTER, POSFILTER,
                                          LENGTHFILTER, EDITDISTANCE, false);
    fullTime = (int) ris.getCompTime();
    pwOutFile2.println("## Seconds used (ApproxStringJoin): "+fullTime);

    Main.MainFrame.setStatusBar("Executing approxStringJoin Pass2 query ...");
    ris = db.approxJoinQueryPass2(TABLE1, TABLE2, QSUB, K, KSUB, LMINSUB,
                                          COUNTFILTER2, true);
    fullTime2 = (int) ris.getCompTime();
    pwOutFile2.println("## Seconds used (ApproxSJ Pass2): "+fullTime2);
   }

int subTime = 0;           // tempo impiegato nella ricerca dei sub match

if (SUBSTR)
    {
     Main.MainFrame.setStatusBar("Executing findSubMatches query...");
     ris = db.findSubMatches (TABLE1, TABLE2, KSUB, LMINSUB);
     subTime = (int) ris.getCompTime();
     pwOutFile2.println("## Seconds used (findSubMatches): "+subTime);
    }

Main.MainFrame.setStatusBar("Executing extractResults query...");
ris = db.extractResults(TABLE1, TABLE2, outputFile2, SUBSTR, vPhrases,
                        DISTDELIMITER, SOFTDELIMITER);

pwOutFile2.println("##\r\n## Total suggestions found: "
                   + ris.getCount());
```

```
    int fullMatches = ((Integer) ris.getResultVector().elementAt(0)).intValue();
    pwOutFile2.println("## FULLMATCH found: "+fullMatches);

    int subMatches = ((Integer) ris.getResultVector().elementAt(1)).intValue();
    if (SUBSTR) pwOutFile2.println("## SUBMATCH found: "+subMatches);

    int totTMPhrases = db.getMaxCod(TABLE1);
    pwOutFile2.println("## Phrases in Translation Memory: "+totTMPhrases);

    int totPhrases = ((Integer) ris.getResultVector().elementAt(2)).intValue();
    pwOutFile2.println("## Phrases to pretranslate: "+totPhrases);

    int fullPhrases = ((Integer) ris.getResultVector().elementAt(3)).intValue();
    pwOutFile2.println("## Phrases found (full): "+fullPhrases);

    int subPhrases = ((Integer) ris.getResultVector().elementAt(4)).intValue();
    if (SUBSTR) pwOutFile2.println("## Phrases found (sub): "+subPhrases);

    pwOutFile2.println("## Seconds used (extractResult): "+ris.getCompTime());

    return new PretranslationStat (fullTime, fullTime2, subTime, totTMPhrases,
                totPhrases, fullPhrases, subPhrases, fullMatches, subMatches);
   }
  catch (SQLException e)
     {
      throw new SQLException("SQL error: "+e.getMessage());
     }
  catch (IOException e)
     {
      throw new IOException("IO error: "+e.getMessage());
     }
  catch (IllegalArgumentException e)
     {
      throw new IllegalArgumentException("Config error: "+e.getMessage());
     }
  finally
     {
      try{
          db.disconnect();                          // disconnessione dal db
         }
      catch(SQLException e){};

      try {outputFile1.close();
           outputFile2.close();
          }             // chiusura file
      catch (IOException e)
      {//System.out.println("Errore in chiusura file: "+e.getMessage());
      };
     }
}
```

```
/**
    Scrive le frasi passate nel file con gli opportuni separatori
*/

private static void writePhrases (Vector vPhrases, FileWriter of)
{
                                    // delimitatori frasi
  final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER;
  final String DISTDELIMITER = Main.MainFrame.DISTDELIMITER;

  PrintWriter pwOutFile = new PrintWriter (of);

  for (int i=0; i<vPhrases.size(); i++)
      {
       String phr = (String) vPhrases.elementAt(i);
       pwOutFile.println(phr);
       pwOutFile.println(DISTDELIMITER+" 0");
       pwOutFile.println(SOFTDELIMITER);
      }
}


/**
  Metodo principale per l'aggiunta di un file alla
  translation memory
*/

 public static String addMemory () throws SQLException, IOException
  {
    final String FILEIN = Main.MainFrame.FILEIN1; // file di input
    final String TABLE1 = Main.MainFrame.TABLE1; // tabella delle frasi di riferimento
    final int Q = Main.MainFrame.Q;                  // numero parole in un qgramma
    final int QSUB = Main.MainFrame.QSUB;
    final boolean STEM = Main.MainFrame.STEM; // attiva la normalizzazione

    FileReader inputFile = null;

    try                         // apertura file di input
     {
      inputFile = new FileReader (FILEIN);
     }
    catch (FileNotFoundException e)
     {
      throw new IOException
              ("Error opening input file : " +e.getMessage());
     }

    Vector vTMPhrases = new Vector();

    try
     {                                  // lettura file di input
      vTMPhrases = GlobalUtility.GlobalUtility.readTMFile (inputFile);
     }
```

```
    catch (IOException e)
     {throw new IOException("Error reading file: "+e.getMessage());
     }
    finally
     {
      try {inputFile.close ();}              // chiusura del file di input
      catch (IOException e) {}
     }

    DBConnect db = null;
    Result ris = null;
    String strRes = new String();

    try                                              // operazioni sul DB
     {
      db = new DBConnect();

      Main.MainFrame.setStatusBar("Extracting and elaborating phrases...");

                                        // il primo codice disponibile
      int startCod = db.getMaxCod(TABLE1) + 1;

      ris = db.memToDB (vTMPhrases, TABLE1, startCod, STEM);

      strRes = strRes + "\n\nPhrases inserted: "+ris.getCount()
                      + "\nSeconds used: "+ris.getCompTime();
      if (STEM) strRes = strRes + "\n(with stemming)";
      else strRes = strRes + "\n(without stemming)";

      Main.MainFrame.setStatusBar("Extracting q-grams...");

      ris = db.fillQTable(TABLE1, startCod, Q, false);
      strRes = strRes + "\nQ-grams inserted: "+ris.getCount()
                      + "\nSeconds used: "+ris.getCompTime();

      if (QSUB != Q)
         {ris = db.fillQTable(TABLE1, startCod, QSUB, true);
          strRes = strRes + "\nQ-grams inserted (sub): "+ris.getCount()
                          + "\nSeconds used: "+ris.getCompTime();
         }

      return strRes;
     }
    catch (SQLException e)
       {
        throw new SQLException("SQL error: "+e.getMessage());
       }
    finally
       {
        try{db.disconnect();}                  // disconnessione dal db
        catch(SQLException e){};
       }
}
```

```
/**
  Controlla lo stato (numero di frasi inserite) della
  tabella della translation memory
*/

 public static int memStatus () throws SQLException
  {
    final String TABLE1 = Main.MainFrame.TABLE1; // tabella delle frasi di riferimento

    DBConnect db = null;

    int num =0;

    try                                    // operazioni sul DB
     {
      db = new DBConnect();
      num = db.getMaxCod(TABLE1);
      return num;                          // numero di frasi nella memoria di traduzione
     }
    catch (SQLException e)
       {
        throw new SQLException("SQL error: "+e.getMessage());
       }
    finally
       {
        try{db.disconnect();}              // disconnessione dal db
        catch(SQLException e){};
       }
  }


/**
  Restituisce la distribuzione statistica delle lunghezze
  delle frasi nella translation memory
*/

 public static int[] memDistr() throws SQLException
  {
    final String TABLE1 = Main.MainFrame.TABLE1; // tabella delle frasi di riferimento

    DBConnect db = null;

    int len [];                            // lunghezze delle frasi

    try                                    // operazioni sul DB
     {
      db = new DBConnect();
      len = db.getLengthDistrib(TABLE1);
      return len;
     }
    catch (SQLException e)
```

```
        {
         throw new SQLException("SQL error: "+e.getMessage());
        }
    finally
        {
         try{db.disconnect();}                  // disconnessione dal db
         catch(SQLException e){};
        }
  }


/**
  Crea tutte le  tabelle
*/

 public static void createTables () throws SQLException
  {
    final String TABLE1 = Main.MainFrame.TABLE1; // tabella delle frasi di riferimento
    final String TABLE2 = Main.MainFrame.TABLE2; // tabella delle frasi di query

    final int Q = Main.MainFrame.Q;                  // numero parole in un qgramma
    final int QSUB = Main.MainFrame.QSUB;
    final boolean SUBSTR = Main.MainFrame.SUBSTR; // ricerca delle sottostringhe

    DBConnect db = null;

    try                                             // operazioni sul DB
     {
      db = new DBConnect();
      db.createPTable(TABLE1);
      db.createQTable(TABLE1, Q, false);
      db.createQTable(TABLE1, QSUB, true);
      db.createPTable(TABLE2);
      db.createQTable(TABLE2, Q, false);
      db.createQTable(TABLE2, QSUB, true);
      db.createResTables();
     }
    catch (SQLException e)
        {
         throw new SQLException("SQL error: "+e.getMessage());
        }
    finally
        {
         try{db.disconnect();}                  // disconnessione dal db
         catch(SQLException e){};
        }
  }


/**
  Elimina tutte le  tabelle
*/
```

```
   public static void dropTables () throws SQLException
    {
      final String TABLE1 = Main.MainFrame.TABLE1; // tabella delle frasi di riferimento
      final String TABLE2 = Main.MainFrame.TABLE2; // tabella delle frasi di query
      final boolean SUBSTR = Main.MainFrame.SUBSTR; // ricerca delle sottostringhe

      DBConnect db = null;

      try                                          // operazioni sul DB
       {
        db = new DBConnect();
        db.dropResTables();
        db.dropQTables(TABLE1);
        db.dropQTables(TABLE2);
        db.dropPTable(TABLE1);
        db.dropPTable(TABLE2);
       }
      catch (SQLException e)
         {
          throw new SQLException("SQL error: "+e.getMessage());
         }
      finally
         {
          try{db.disconnect();}                   // disconnessione dal db
          catch(SQLException e){};
         }
    }
}
```

A.1.8 Classe SimSearch.Utility

```
package SimSearch;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                          Utility
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
      Classe astratta contenente funzioni di utilita'
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;
import java.io.*;
import Stemming.eng.*;

public abstract class Utility
{
   /**
     Restituisce un vettore di q-grammi dalla stringa in ingresso
     (la dimensione q dei q-grammi e' un parametro della funzione)
   */
```

```
public static Vector qgramVector (String s, int q, boolean extend)
 {
  String extendedString = new String();

                                    // estensione stringa con caratteri # e $
                                    // (se extend == true)
  if (extend == true)
     {extendedString = extendString (s, q);}
  else
     {extendedString = s;}

  Vector vQgram = new Vector();    // vettore di q−grammi del risultato

                           // indici che definiscono la finestra che scorre
                           // sulla stringa per formare i qgrammi
  int inizio =−1;
  int fine =0;

  for (int i=0; i<q; i++)
      {
              // posizionamento dell'indice fine alla fine del primo qgramma
       fine = extendedString.indexOf(" ", fine+1);
      }

  int pos=1;          // contatore posizione qgramma corrente

  while (fine!=−1)          // ciclo di aggiunta qgrammi al vettore risultato
      {
       vQgram.addElement
              (new PosQgram (pos, extendedString.substring(inizio+1, fine)));
       pos++;
                                          // avanzamento della finestra
       inizio = extendedString.indexOf(" ", inizio +1);
       fine = extendedString.indexOf(" ", fine+1);
      }

                                                  // aggiunta qgramma finale
  vQgram.addElement (new PosQgram (pos, extendedString.substring(inizio+1)));

  return vQgram;
 }


/**
   Restituisce la frase priva di caratteri di punteggiatura e speciali
   ( utilizzata in caso di normalizzazione disattivata )
*/

public static Output cleanPhrase (String s)
{
  StringTokenizer st = new StringTokenizer(s);
```

```
  String token;
  String phr = "";
  String pos = "";

  int i=1;

  while (st.hasMoreTokens())
      {
       token = st.nextToken();
                              // token di punteggiatura
       if (token.equals("." ) ||  token.equals(";" ) ||
           token.equals(":" ) ||  token.equals("," ) ||
           token.equals("?" ) ||  token.equals("!" ) ||
           token.equals("(" ) ||  token.equals(")" ) ||
           token.equals("\""))
           {}
       else
           {phr = phr + token + " ";
            pos = pos + "<" + i + ">";
           }
       i++;
      }
  return new Output((long) 0, phr, pos);    // numero parole di s
}


/**
   Normalizza una frase (se stem = true)
   oppure la ripulisce solo della punteggiatura (se stem = false)
*/

public static Output elabPhrase (String fraseSorg, boolean stem)
 {
  if (stem)                // normalizzazione
     {
      Output stemOutput = NormalizzaFrase.normalizz(fraseSorg);
      return stemOutput;
     }
  else                     // rimozione punteggiatura
     {
      Output noStemOutput = cleanPhrase(fraseSorg);
      return noStemOutput;
     }
 }


/**
  Estende la stringa data in ingresso per la
  costruzione dei q-grammi
  es con q=3:  "AA BB" -> "# # AA BB $ $"
*/

public static String extendString (String s , int q)
```

```
  {
    String extString = new String ();

    for (int i=1; i<q; i++)
        {
          extString = extString + "# ";          // caratteri iniziali
        }

    extString = extString + s;

    for (int i=1; i<q; i++)
        {
          extString = extString + " $";          // caratteri finali
        }

    return extString;
  }
}
```

A.2 Package Align

A.2.1 Classe Align.SentAlign

```
package Align;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                        SentAlign
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
      Classe che realizza l'allineamento delle frasi
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;
import java.io.*;

public abstract class SentAlign
{
  /**
      Funzione principale di allineamento frasi
  */

  public static void sentAlignMain () throws IOException
  {
   final String HARDDELIMITER = Main.MainFrame.HARDDELIMITER; // delimitatori
   final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER;
   final String FILEIN1 = Main.MainFrame.FILEIN1; // file di input
   final String FILEIN2 = Main.MainFrame.FILEIN2;
```

```
FileReader inputFile1 = null;              // apertura dei file di input
FileReader inputFile2 = null;

try
  {
   inputFile1 = new FileReader (FILEIN1);
   inputFile2 = new FileReader (FILEIN2);
  }
catch (FileNotFoundException e)
  {
   throw new IOException ("Error opening input file: "+e.getMessage());
  }

String fileout1 = FILEIN1 + ".al";       // apertura dei file di output
String fileout2 = FILEIN2 + ".al";
FileWriter outputFile1 = null;
FileWriter outputFile2 = null;

try
  {
   outputFile1 = new FileWriter (fileout1);
   outputFile2 = new FileWriter (fileout2);
  }
catch (IOException e)
  {
   throw new IOException ("Error creating output file: "+e.getMessage());
  }

Vector vLines1 = new Vector();             // lettura dei file di input
Vector vLines2 = new Vector();

try
  {
   vLines1 = GlobalUtility.GlobalUtility.readLines(inputFile1);
   vLines2 = GlobalUtility.GlobalUtility.readLines(inputFile2);
  }
catch (IOException e)
  {
   throw new IOException ("Error reading file: "+e.getMessage());
  }
finally
  {
    try
     {
      inputFile1.close();
      inputFile2.close();
     }
    catch (IOException e) {}
  }

// controllo sugli HARD DELIMITER e separazione paragrafi

Vector hardRegions1 = findSubRegions(vLines1, HARDDELIMITER);
```

```
int numberOfHardRegions1 = hardRegions1.size();

Vector hardRegions2 = findSubRegions(vLines2, HARDDELIMITER);
int numberOfHardRegions2 = hardRegions2.size();

if(numberOfHardRegions1 != numberOfHardRegions2)
   {
    throw new IOException
            ("Input files don't contain the same number of hard delimiters");
   }

// ciclo di allinemento frasi (per ciascun paragrafo)

for(int i=0; i<numberOfHardRegions1; i++)
   {
    Main.MainFrame.setStatusBar("Computing Sentece Align ("+i
                                +" of "+(numberOfHardRegions1)+") ...");

                                // separazione frasi in paragrafi
    Vector softRegions1 = findSubRegions((Vector) hardRegions1.elementAt(i), SOFTDELIMITER);
    Vector softRegions2 = findSubRegions((Vector) hardRegions2.elementAt(i), SOFTDELIMITER);

    int len1[] = regionLengths(softRegions1);     // array delle lunghezze da allineare
    int len2[] = regionLengths(softRegions2);

    Vector align = sAlign(len1, len2);      // l'allineamento vero e proprio
                                            // (restituisce un vettore di oggetti SentAlignment)

                                            // output dei file allineati
    writeSentAlign (align, outputFile1, outputFile2, softRegions1, softRegions2);
   }

try {                                       // chiusura file
     outputFile1.close();
     outputFile2.close();
    }
catch (IOException e)
  {throw new IOException("Error closing file: "+e.getMessage());
  }
}


/**
   Scrive i file con le frasi allineate
*/

private static void writeSentAlign (Vector align, FileWriter of1, FileWriter of2,
                                    Vector softRegions1, Vector softRegions2)
{
 final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER;
 final String DISTDELIMITER = Main.MainFrame.DISTDELIMITER;

 PrintWriter pwOutFile1 = new PrintWriter (of1);
```

```
  PrintWriter pwOutFile2 = new PrintWriter (of2);

  int ix = 0; int iy = 0;
  int prevx = 0; int prevy = 0;

  for (int j = 0; j < align.size (); j++)
    {
     SentAlignment a = (SentAlignment) align.elementAt(j);

     if(a.getx2() > 0) ix++;
     else if(a.getx1() == 0) ix--;
     if(a.gety2() > 0) iy++;
     else if(a.gety1() == 0) iy--;
     if(a.getx1() == 0 && a.gety1() == 0 && a.getx2() == 0 && a.gety2() == 0)
         {ix++; iy++;}

     ix++; iy++;

     for(; prevx < ix; prevx++)
        printRegion(pwOutFile1, (Vector) softRegions1.elementAt(prevx));
     pwOutFile1.println(DISTDELIMITER+" "+a.getd());
     pwOutFile1.println(SOFTDELIMITER);

     for(; prevy < iy; prevy++)
        printRegion(pwOutFile2, (Vector) softRegions2.elementAt(prevy));
     pwOutFile2.println(DISTDELIMITER+" "+a.getd());
     pwOutFile2.println(SOFTDELIMITER);
    }
}


/**
    Funzione di preparazione all'allineamento delle frasi
    (con riconoscimento e separazione frasi e paragrafi)
*/

public static int prepareToAlign (boolean oneLine) throws IOException
{
 final String FILEIN1 = Main.MainFrame.FILEIN1; // file di input
 final String FILEIN2 = Main.MainFrame.FILEIN2;

 // apertura dei file di input e output

 FileReader inputFile1 = null;
 FileReader inputFile2 = null;

 try
   {
    inputFile1 = new FileReader (FILEIN1);
    inputFile2 = new FileReader (FILEIN2);
   }
 catch (FileNotFoundException e)
   {
```

```
    throw new IOException
          ("Error opening input file : " +e.getMessage());
  }

String fileout1 = FILEIN1 + ".prep";
String fileout2 = FILEIN2 + ".prep";
FileWriter outputFile1 = null;
FileWriter outputFile2 = null;

try
  {
   outputFile1 = new FileWriter (fileout1);
   outputFile2 = new FileWriter (fileout2);
  }
catch (IOException e)
  {
   throw new IOException
         ("Error creating output file : " +e.getMessage());
  }

// lettura dei file di input

Vector vPhrase1 = new Vector();
Vector vPhrase2 = new Vector();

try
  {                        // riconoscimento frasi e paragrafi
   if (!oneLine)
      {vPhrase1 = GlobalUtility.GlobalUtility.readPhrases (inputFile1);
       vPhrase2 = GlobalUtility.GlobalUtility.readPhrases (inputFile2);
      }
   else                    // frasi gi segmentate
      {vPhrase1 = GlobalUtility.GlobalUtility.readLines (inputFile1);
       vPhrase2 = GlobalUtility.GlobalUtility.readLines (inputFile2);
      }
  }
catch (IOException e)
  {
   throw new IOException ("Error reading file: "+e.getMessage());
  }
finally
  {
    try
     {
      inputFile1.close ();
      inputFile2.close ();
     }
    catch (IOException e) {}
  }

                  // scrittura dei file preparati per l'allineamento
int hardCount1 = writePrepFile (vPhrase1, outputFile1);
int hardCount2 = writePrepFile (vPhrase2, outputFile2);
```

```
 try {                                  // chiusura file
       outputFile1.close ();
       outputFile2.close ();
      }
 catch (IOException e)
   {throw new IOException("Error closing file: "+e.getMessage());
   }

 if (hardCount1 == hardCount2) return (0);
 else return (-1);
 }


/**
    Scrive il file con le frasi preparate per l'allineamento
*/

private static int writePrepFile (Vector vPhrase, FileWriter of)
{
 final String HARDDELIMITER = Main.MainFrame.HARDDELIMITER; // delimitatori
 final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER;
 final int WORDSIZE = Main.MainFrame.WORDSIZE; // le frasi con parole pi lunghe di WORDSIZE
                                               // saranno scartate

 int hardCount = 0;       // conteggio paragrafi nel file

 PrintWriter pwOutFile = new PrintWriter (of);

 for (int i=0; i<vPhrase.size(); i++)
    {
     String phr = (String) vPhrase.elementAt(i);

     if (! wordSizeOK(phr, WORDSIZE)) continue; // frase scartata: parole troppo lunghe

     if (! phr.equals("") )        // nuova frase
        {
         pwOutFile.println(phr);
         pwOutFile.println(SOFTDELIMITER);
        }
     else                          // nuovo paragrafo
        {
         if ( i != vPhrase.size()-1)
                pwOutFile.println(HARDDELIMITER);
         hardCount++;
        }
    }

 pwOutFile.println(HARDDELIMITER);

 return (hardCount);
}
```

```
/**
    Funzione di preparazione all'allineamento delle frasi
    per files con frasi gi allineate (separate da tabulazione)
*/

public static void prepareTabbed () throws IOException
{
 final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER; // delimitatori
 final String DISTDELIMITER = Main.MainFrame.DISTDELIMITER;

 final String FILEIN = Main.MainFrame.FILEIN1; // file di input

 final int WORDSIZE = Main.MainFrame.WORDSIZE;

 // apertura dei file di input e output

 FileReader inputFile = null;
 String fileout1 = FILEIN + "1.al";
 String fileout2 = FILEIN + "2.al";
 FileWriter outputFile1 = null;
 FileWriter outputFile2 = null;

 try
   {
    inputFile = new FileReader (FILEIN);
   }
 catch (FileNotFoundException e)
   {
    throw new IOException ("Error opening input file: "+e.getMessage());
   }

 try
   {
    outputFile1 = new FileWriter (fileout1);
    outputFile2 = new FileWriter (fileout2);
   }
 catch (IOException e)
   {
    throw new IOException ("Error creating output file: "+e.getMessage());
   }

 // lettura dei file di input

 Vector vLines = new Vector();

 try
   {
    vLines = GlobalUtility.GlobalUtility.readLines(inputFile);
   }
 catch (IOException e)
   {
    throw new IOException ("Error reading input file: "+e.getMessage());
```

```
    }
  finally
    {
     inputFile.close();
    }

  PrintWriter pwOutFile1 = new PrintWriter (outputFile1);
  PrintWriter pwOutFile2 = new PrintWriter (outputFile2);

  for (int i=0; i<vLines.size(); i++)
     {
      String line = (String) vLines.elementAt(i);

      if (! wordSizeOK(line, WORDSIZE)) continue; // linea scartata: parole troppo lunghe

      int tabPos1 = line.indexOf('\t');
      if (tabPos1 < 0) continue;                  // linea scartata: nessun tab
      int tabPos2 = line.indexOf('\t', tabPos1+1);
      if (tabPos2 >= 0) continue;                 // linea scartata: pi di un tab

                                                  // linea OK
      pwOutFile1.println(line.substring(0,tabPos1));
      pwOutFile1.println(DISTDELIMITER+" 0");
      pwOutFile1.println(SOFTDELIMITER);

      pwOutFile2.println(line.substring(tabPos1+1));
      pwOutFile2.println(DISTDELIMITER+" 0");
      pwOutFile2.println(SOFTDELIMITER);
     }

  try {                                 // chiusura file
       outputFile1.close();
       outputFile2.close();
      }
  catch (IOException e)
    {//System.out.println("Errore in chiusura file: "+e.getMessage());
    };
}

/**
   Metodo che verifica se le parole della frase sono pi
   lunghe di wordSize
*/

private static boolean wordSizeOK(String s, int wordSize)
 {
  StringTokenizer stok = new StringTokenizer(s);

  while (stok.hasMoreTokens())
   {
     if (stok.nextToken().length() >= wordSize) return false;
   }
  return true;
```

```
}

/**
  Metodo che esegue l'allineamento vero e proprio
  x e y sono sequenze di oggetti, rappresentati da interi non nulli,
  da allineare
*/

private static Vector sAlign (int x [], int y [])
{
 final int MAXINT = 1000000;

 int nx = x.length;
 int ny = y.length;

 int dist [][] = new int[nx+1][ny+1];       // array delle distanze di allineamento
 int pathx [][] = new int[nx+1][ny+1];      // array del percorso di allineamento
 int pathy [][] = new int[nx+1][ny+1];
 int d1, d2, d3, d4, d5, d6, dmin;

 for(int j = 0; j <= ny; j++)
   {
    for(int i = 0; i <= nx; i++)
      {
       d1 = i>0 && j>0 ?  // sostituzione
       dist[i-1][j-1] + twoSideDistance(x[i-1], y[j-1], 0, 0)
       : MAXINT;
       d2 = i>0 ?             // cancellazione
       dist[i-1][j] + twoSideDistance(x[i-1], 0, 0, 0)
       : MAXINT;
       d3 = j>0 ?             // inserimento
       dist[i][j-1] + twoSideDistance(0, y[j-1], 0, 0)
       : MAXINT;
       d4 = i>1 && j>0 ?  // contrazione
       dist[i-2][j-1] + twoSideDistance(x[i-2], y[j-1], x[i-1], 0)
       : MAXINT;
       d5 = i>0 && j>1 ?  // espansione
       dist[i-1][j-2] + twoSideDistance(x[i-1], y[j-2], 0, y[j-1])
       : MAXINT;
       d6 = i>1 && j>1 ?  // fusione
       dist[i-2][j-2] + twoSideDistance(x[i-2], y[j-2], x[i-1], y[j-1])
       : MAXINT;

       dmin = d1;
       if(d2<dmin) dmin=d2;
       if(d3<dmin) dmin=d3;
       if(d4<dmin) dmin=d4;
       if(d5<dmin) dmin=d5;
       if(d6<dmin) dmin=d6;

       if(dmin == MAXINT)
         {
          dist[i][j] = 0;
```

```
        }
      else if(dmin == d1)     // sostituzione
        {
         dist[i][j] = d1;  pathx[i][j] = i-1;  pathy[i][j] = j-1;
        }
      else if(dmin == d2)     // cancellazione
        {
         dist[i][j] = d2;  pathx[i][j] = i-1;  pathy[i][j] = j;
        }
      else if(dmin == d3)     // inserimento
        {
         dist[i][j] = d3;  pathx[i][j] = i;  pathy[i][j] = j-1;
        }
      else if(dmin == d4)     // contrazione
        {
         dist[i][j] = d4;  pathx[i][j] = i-2;  pathy[i][j] = j-1;
        }
      else if(dmin == d5)     // espansione
        {
         dist[i][j] = d5;  pathx[i][j] = i-1;  pathy[i][j] = j-2;
        }
      else  // dmin == d6     // fusione
        {
         dist[i][j] = d6;  pathx[i][j] = i-2;  pathy[i][j] = j-2;
        }
     }
  }

Vector align = new Vector();
Vector ralign = new Vector();

int oi, oj, di, dj;

for(int i=nx, j=ny ; i>0 || j>0 ; i = oi, j = oj)
  {
   oi = pathx[i][j];
   oj = pathy[i][j];
   di = i - oi;
   dj = j - oj;
   if(di == 1 && dj == 1)
     { // sostituzione
      SentAlignment alignS =
          new SentAlignment(x[i-1],y[j-1],0,0,dist[i][j] - dist[i-1][j-1]);
      ralign.addElement(alignS);
     }
   else if(di == 1 && dj == 0)
     { // cancellazione
      SentAlignment alignS =
          new SentAlignment(x[i-1],0,0,0,dist[i][j] - dist[i-1][j]);
      ralign.addElement(alignS);
     }
   else if(di == 0 && dj == 1)
     { // inserimento
```

```
        SentAlignment alignS =
            new SentAlignment(0,y[j-1],0,0,dist[i][j] - dist[i][j-1]);
        ralign.addElement(alignS);
       }
     else if(dj == 1)
       { // contrazione
        SentAlignment alignS =
            new SentAlignment(x[i-2],y[j-1],x[i-1],0,dist[i][j] - dist[i-2][j-1]);
        ralign.addElement(alignS);
       }
     else if(di == 1)
       { // espansione
        SentAlignment alignS =
            new SentAlignment(x[i-1],y[j-2],0,y[j-1],dist[i][j] - dist[i-1][j-2]);
        ralign.addElement(alignS);
       }
     else // di == 2 && dj == 2
       { // fusione
        SentAlignment alignS =
            new SentAlignment(x[i-2],y[j-2],x[i-1],y[j-1],dist[i][j] - dist[i-2][j-2]);
        ralign.addElement(alignS);
       }
    }

  for (int i = ralign.size()-1; i>=0; i--)
     {
      align.addElement ((SentAlignment) ralign.elementAt(i));
     }
  return(align);
}


/**
   Restituisce l'area sottesa da una distribuzione normale
   compresa tra -inf e z (deviazione standard)
*/

private static double pnorm (double z)
 {
  double t, pd;
  t = 1/(1 + 0.2316419 * z);
  pd = 1 - 0.3989423 * Math.exp(-z * z/2) * ((((1.330274429 * t - 1.821255978) * t
        + 1.781477937) * t - 0.356563782) * t + 0.319381530) * t;
        /* vedere Abramowitz, M., and I. Stegun (1964), 26.2.17 p. 932 */
  return(pd);
 }


/**
   Restituisce -100 * log della probabilit che una frase di lunghezza
   len1 sia la traduzione di una frase di lunghezza len2. La
   probabilit basata su due parameri, la media e la varianza del
   numero di caratteri stranieri per carattere della lingua di partenza
```

```
*/

private static int match(int len1, int len2)
 {
                    // parametri per l'allineamento delle lunghezze
  final double C = Main.MainFrame.C;
  final double S2 = Main.MainFrame.S2;

  final int BIGDISTANCE = 2500;

  double z, pd, mean;
  if(len1==0 && len2==0) return(0);
  mean = (len1 + len2/C)/2;
  z = (C * len1 - len2)/Math.sqrt(S2 * mean);
          // Si utilizzano entrambi i lati della distribuzione binomiale
  if(z < 0) z = -z;
  pd = 2 * (1 - pnorm(z));
  if(pd > 0) return((int)(-100 * Math.log(pd)));
  else return(BIGDISTANCE);
}


/**
  Funzione di distanza, con 4 argomenti:

    twoSideDistance(x1, y1, 0, 0) d il costo di sostituzione of x1 by y1
    twoSideDistance(x1, 0, 0, 0) d il costo di cancellazione of x1
    twoSideDistance(0, y1, 0, 0) d il costo di inserimento of y1
    twoSideDistance(x1, y1, x2, 0) d il costo di contrazione of (x1,x2) to y1
    twoSideDistance(x1, y1, 0, y2) d il costo di espansione of x1 to (y1,y2)
    twoSideDistance(x1, y1, x2, y2) d il costo di fusione da (x1,x2) a (y1,y2)
*/

private static int twoSideDistance (int x1, int y1, int x2, int y2)
 {
  final int PENALTY21 = 230; // -100 * log([prob di match 2-1] / [prob di match 1-1])
  final int PENALTY22 = 440; // -100 * log([prob di match 2-2] / [prob di match 1-1])
  final int PENALTY01 = 450; // -100 * log([prob di match 0-1] / [prob di match 1-1])

  if(x2 == 0 && y2 == 0)
  if(x1 == 0)      // inserimento
  return(match(x1, y1) + PENALTY01);
  else if(y1 == 0) // cancellazione
  return(match(x1, y1) + PENALTY01);
  else return (match(x1, y1));    // sostituzione
  else if(x2 == 0) // espansione
  return (match(x1, y1 + y2) + PENALTY21);
  else if(y2 == 0) // contrazione
  return(match(x1 + x2, y1) + PENALTY21);
  else             // fusione
  return(match(x1 + x2, y1 + y2) + PENALTY22);
 }
```

```
/**
  Stampa una regione
*/

private static void printRegion (PrintWriter pw, Vector region)
 {
  for( int i=0; i<region.size () ; i++)
    pw.println((String) region.elementAt(i));
 }


/**
  Restituisce la lunghezza in caratteri di una regione
*/

private static int lengthOfARegion (Vector region)
 {
  int length = 0;

  for ( int i=0; i<region.size (); i++)
     length += ((String) region.elementAt(i)).length ();
  return (length);
 }


/**
  Restituisce le lunghezze in caratteri delle sottoregioni
  contenute nella regione passata come parametro
*/

private static int[] regionLengths (Vector regions)
 {
  int n = regions.size ();
  int lengths[] = new int[n];

  for(int i = 0; i < n; i++)
  lengths[i] = ((int) lengthOfARegion((Vector) regions.elementAt(i)));

  return(lengths);
}


/**
  Restituisce un vettore contenente le sottoregioni trovate nella
  regione passata come parametro (il delimitatore quello specificato )
*/

private static Vector findSubRegions(Vector vRegion, String delimiter)
 {
  Vector vSubRegion = new Vector();
  Vector vResult = new Vector();
```

```
    for (int i=0; i<vRegion.size(); i++)
      {
       if (((String) vRegion.elementAt(i)).equals (delimiter))
         {
          vResult.addElement((Vector) vSubRegion);
          vSubRegion = new Vector();
         }
       else vSubRegion.addElement ((String) vRegion.elementAt(i));

      }
    return(vResult);
   }
}
```

A.2.2 Classe Align.SentAlignment

```
package Align;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                        SentAlignment
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
    Classe che implementa l'oggetto allinemento tra frasi
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

public class SentAlignment
{
  private int x1;  private int y1;
  private int x2;  private int y2;
  private int d;

  /**
    Costruttore
  */

  SentAlignment (int x1i, int y1i , int x2i , int y2i , int di)
   {
    x1 = x1i;  y1 = y1i;
    x2 = x2i;  y2 = y2i;
    d = di;
   }

  /**
     Restituisce x1
  */

  public int getx1 ()
   {
     return x1;
   }
```

```
  /**
     Restituisce y1
  */

  public int gety1 ()
   {
     return y1;
   }

  /**
     Restituisce x2
  */

  public int getx2 ()
   {
     return x2;
   }

  /**
     Restituisce y2
  */

  public int gety2 ()
   {
     return y2;
   }

  /**
     Restituisce la distanza
  */

  public int getd ()
   {
     return d;
   }
}
```

A.2.3 Classe Align.Token

```
package Align;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                          Token
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
        Classe che implementa l'oggetto Token
        (parola o simbolo di punteggiatura e relativa
        posizione all'interno della frase)
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/
```

```
public class Token
{
  private int pos;
  private String token;


  /**
    Costruttore
  */

  Token (int p, String s)
   {
    pos = p;
    token = s;
   }

  /**
     Restituisce la posizione del token
  */

  public int getPos ()
   {
     return pos;
   }


  /**
     Restituisce la stringa del token
  */

  public String getToken ()
   {
     return token;
   }
}
```

A.2.4 Classe Align.WordAlign

```
package Align;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                          WordAlign
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
       Classe che realizza l'allineamento delle parole
       nelle frasi
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;
```

```
import java.io.*;

public abstract class WordAlign
{
  /**
      Funzione principale di allineamento parole
  */

  public static void wordAlignMain () throws IOException
  {
   final String SOFTDELIMITER = Main.MainFrame.SOFTDELIMITER; // delimitatori
   final String DISTDELIMITER = Main.MainFrame.DISTDELIMITER;

   final String FILEIN1 = Main.MainFrame.FILEIN1; // file di input
   final String FILEIN2 = Main.MainFrame.FILEIN2;

   FileReader inputFile1 = null;          // apertura dei file di input
   FileReader inputFile2 = null;

   try
     {
      inputFile1 = new FileReader (FILEIN1);
      inputFile2 = new FileReader (FILEIN2);
     }
   catch (FileNotFoundException e)
     {
      throw new IOException
              ("Error opening input file : " +e.getMessage());
     }

   String fileout = FILEIN1 + ".tm";    // apertura dei file di output
   FileWriter outputFile = null;

   try
     {
      outputFile = new FileWriter (fileout);
     }
   catch (IOException e)
     {
      throw new IOException
             ("Error creating output file : " +e.getMessage());
     }

   // lettura dei file di input

   Vector vLines1 = new Vector();
   Vector vLines2 = new Vector();

   Main.MainFrame.setStatusBar("Computing Word Align (reading files) ...");

   try
     {
      vLines1 = GlobalUtility.GlobalUtility.readLines(inputFile1);
```

```
    vLines2 = GlobalUtility.GlobalUtility.readLines(inputFile2);
  }
catch (IOException e)
  {
    throw new IOException ("Error reading file: "+e.getMessage());
  }
finally
  {
    try
     {
      inputFile1.close ();
      inputFile2.close ();
     }
    catch (IOException e) {}
  }

Main.MainFrame.setStatusBar("Computing Word Align (removing comments) ...");
vLines1 = removeComments (vLines1);     // rimozione eventuali linee di commento (##)
vLines2 = removeComments (vLines2);

Main.MainFrame.setStatusBar("Computing Word Align (extracting distances) ...");
Vector vDist = extractDist (vLines1, DISTDELIMITER); // estrazione e rimozione informazioni
vLines1 = removeDist (vLines1, DISTDELIMITER);      // di distanza
vLines2 = removeDist (vLines2, DISTDELIMITER);

Main.MainFrame.setStatusBar("Computing Word Align (preparing phrases) ...");
Vector vSent1 = mergeRegions(vLines1, SOFTDELIMITER); // estrazione (ed eventuale fusione)
Vector vSent2 = mergeRegions(vLines2, SOFTDELIMITER); // delle frasi in vettore

int regNum1 = vSent1.size();                // controllo stesso numero di frasi
int regNum2 = vSent2.size();

if (regNum1 != regNum2)
   {
    throw new IOException
            ("Input files don't contain the same number of soft delimiters");
   }

Main.MainFrame.setStatusBar("Computing Word Align (separating punctuation) ...");
                                                  // separazione punteggiatura dalle frasi
vSent1 = GlobalUtility.GlobalUtility.separatePunct (vSent1);
vSent2 = GlobalUtility.GlobalUtility.separatePunct (vSent2);

Main.MainFrame.setStatusBar("Computing Word Align (cleaning spaces) ...");
vSent1 = GlobalUtility.GlobalUtility.cleanSpaces (vSent1);      // eliminazione doppi spazi
vSent2 = GlobalUtility.GlobalUtility.cleanSpaces (vSent2);

wAlign (vSent1, vSent2, vDist, outputFile);      // esecuzione e scrittura allineamento parole

try {                                  // chiusura file
     outputFile.close ();
    }
catch (IOException e)
```

```
    {throw new IOException("Error closing file: "+e.getMessage());
    };
}


/**
  Esegue l'allineamento delle parole e lo scrive nel file di output
*/

private static void wAlign (Vector vSent1, Vector vSent2, Vector vDist, FileWriter outputFile)
{
 PrintWriter pwOutFile = new PrintWriter (outputFile);

                                    // ciclo di allineamento per ogni coppia di regioni (frasi)
 for (int i=0; i<vSent1.size(); i++)
    {
     Main.MainFrame.setStatusBar("Computing Word Align ("+i+" of "+(vSent1.size())+") ...");

     Vector vTok1 = findTokens ((String) vSent1.elementAt(i));
     Vector vTok2 = findTokens ((String) vSent2.elementAt(i));

                                // numero totale di token nelle regioni (frasi) correnti
     int regLen1 = ((Integer) vTok1.elementAt(3)).intValue();
     int regLen2 = ((Integer) vTok2.elementAt(3)).intValue();

     WordAlignment wa = new WordAlignment (regLen1, regLen2);

                                        // allineamento punteggiatura
     wa = tokAlign ((Vector) vTok1.elementAt(0), (Vector) vTok2.elementAt(0), wa, false);
                                        // allinemento token numerici
     wa = tokAlign ((Vector) vTok1.elementAt(1), (Vector) vTok2.elementAt(1), wa, false);
                                        // allinemento parole selezionate
     wa = tokAlign ((Vector) vTok1.elementAt(2), (Vector) vTok2.elementAt(2), wa, true);

     wa = validateAlign (wa);          // completamento e controllo allineamento

     int pos[] = wa.getPos();          // posizioni dell'allineamento delle parole
     int score[] = wa.getScore();      // punteggi dell'allineamento delle parole
                                       // distanza dell'allineamento delle frasi
     int dist = ((Integer) vDist.elementAt(i)).intValue();

     String posString = new String();    // costruzione stringhe di posizione e punteggio
     String scoreString = dist + " ";

     for (int al=0; al<pos.length; al++)
        {
         posString = posString + "<" + pos[al] + ">";
         scoreString = scoreString + "<" + score[al] + ">";
        }
                                        // scrittura del file di output
     pwOutFile.println((String) vSent1.elementAt(i));
     pwOutFile.println((String) vSent2.elementAt(i));
     pwOutFile.println(posString);
```

```
      pwOutFile.println(scoreString);
     }
}


/**
  Allinea i token passati come argomento
*/

 private static WordAlignment tokAlign (Vector vTok1, Vector vTok2,
                                        WordAlignment wa, boolean lcs)
  {
   final int MINSCORE = Main.MainFrame.MINSCORE; // minimo punteggio per considerare due
                                                 // parole allineate

   int pos[] = wa.getPos();                 // array di posizioni di allineamento
   int score[] = wa.getScore();             // array di punteggi di allineamento
   int targetLen = wa.getTargetLen();       // lunghezza della frase target

   if (vTok1.size() == 0 || vTok2.size() == 0)      // nessun token passato
       return wa;

                                                 // rapporto tra le lunghezze delle due frasi
   double lenRatio = (double) pos.length / targetLen;

                                    // ciclo allineamento token
   for (int j=0; j<vTok2.size(); j++)
     {
      Token tok2 = (Token) vTok2.elementAt(j);

      int bestScore = MINSCORE; // migliore punteggio di allineamento per il token corrente
      int bestPos = -1;             // corrispondente posizione di allineamento

                                    // ricerca miglior allineamento per il token corrente
      for (int i=0; i< vTok1.size(); i++)
        {
         Token tok1 = (Token) vTok1.elementAt(i);

         int curScore = matchScore(tok1, tok2, lenRatio, lcs);

         if (curScore > score[tok1.getPos()] && curScore > bestScore)
             {
              bestScore = curScore;
              bestPos = tok1.getPos();
             }
        }

      if (bestPos != -1)         // trovato un possibile allineamento
        {
         pos[bestPos] = tok2.getPos()+1;
         score[bestPos] = bestScore;
        }
     }
```

```
  return new WordAlignment (pos, score, targetLen);
 }


/**
   Restituisce un punteggio riguardante l'allineamento dei
   due token passati come parametro:
   valori pi alti corrispondono ad accoppiamenti pi probabili
*/

private static int matchScore (Token t1, Token t2, double lr, boolean lcs)
 {
  final int EQMULT = Main.MainFrame.EQMULT; // punteggio di all.: parole uguali
  final int LCSMULT = Main.MainFrame.LCSMULT; // punteggio di all.: parole simili
  final int DICMULT = Main.MainFrame.DICMULT; // punteggio di all.: parole da dizionario

  final double MINLCS = Main.MainFrame.MINLCS; // soglia minima di somiglianza

  String s1 = t1.getToken();
  String s2 = t2.getToken();

  int pos1 = t1.getPos();
  int pos2 = t2.getPos();

  double pos2Rel = (pos2*lr);

                                          // punteggio di uguaglianza
  if (s1.equalsIgnoreCase(s2)) return (int) (EQMULT * decay(pos1, pos2Rel));
  else if ( lcs )
    {
     double lcsScore = lcs(s1,s2 ) / (( double) Math.max(s1.length(), (int) s2.length()));
                                          // punteggio di somiglianza
     if ( lcsScore >= MINLCS) return (int) (LCSMULT * lcsScore * decay(pos1, pos2Rel));
     else return 0;
    }
  else return 0;
 }


/**
   Restituisce il coefficiente di smorzamento
   (maggiore quanto pi grande la differenza tra le posizioni passate come parametro)
*/

private static double decay (int p1, double p2)
 {
                          // parametri per definire la funzione di smorzamento
  final double COEFF = Main.MainFrame.COEFF;
  final double MINDECAY = Main.MainFrame.MINDECAY;

  double decay = MINDECAY + (1 - MINDECAY) / ( 1 + COEFF * Math.pow(p1-p2,2));
  return decay;
```

```
  }


/**
  Controlla (eventualmente correggendo)
  e completa l'allineamento mediante interpolazione
*/

 private static WordAlignment validateAlign (WordAlignment wa)
  {
     // parametri per l'eliminazione di picchi (tipicamente errati) nell'allineamento
   final double MAXSLRATIO = Main.MainFrame.MAXSLRATIO;
   final double MAXSCRATIO = Main.MainFrame.MAXSCRATIO;

   int pos[] = wa.getPos();
   int score[] = wa.getScore();
   int targetLen = wa.getTargetLen();

   if (pos[0] == -1)                  // eventuale aggiunta allineamento parole iniziali
      {
       if (checkReverse(pos,1,pos.length-1))
          pos[0] = targetLen;              // inversione allineamento
       else pos[0] = 1;                    // allineamento standard
       score[0] = 1;
      }

   if (pos[pos.length -1] == -1)    // eventuale aggiunta allineamento parole finali
      {
       if (checkReverse(pos,targetLen,0))
           pos[pos.length-1] = 1;                 // inversione allineamento
       else pos[pos.length -1] = targetLen;       // allineamento standard
       score[pos.length -1] = 1;
      }

   int start = -1;
   int end = 0;
   int end2 = -1;

                  // ciclo di controllo e interpolazione
   while (end != -1)
      {
       if (end2 != -1)
         {
          double sl1 = Math.abs ((double) (pos[end]-pos[start]) / (end-start));
          double sl2 = Math.abs ((double) (pos[end2]-pos[start]) / (end2-start));

                                                 // eliminazione eventuale picco errato
          if (sl1 > sl2 * MAXSLRATIO && score[end2] > score[end] * MAXSCRATIO)
             {
              score[end] = -1;
              end = end2;
             }
         }
```

```
        for (int i=start+1; i<end; i++)          // interpolazione
          {
           pos[i]= pos[start] + ((pos[end]-pos[start]) * (i-start)) / (end-start);
          }

        start = nextMatch (score, start);
        if (start == -1) break;

        end = nextMatch (score, start);
        if (end == -1) break;

        end2 = nextMatch (score, end);
       }

    return new WordAlignment (pos, score, targetLen);
  }


/**
   Dato il vettore delle posizioni, verifica se necessario
   invertire l'allineamento delle parole iniziali e finali
*/

private static boolean checkReverse (int pos[], int val, int oppositePos)
  {
    if (pos[oppositePos] == val)
       {
        for (int i=0; i<pos.length; i++)
            if (pos[i] != -1 && pos[i] != val)
               {return false;}
        return true;
       }
     else return false;
  }

/**
   Dato il vettore di punteggi, restituisce la posizione
   del match che segue quella passata come parametro
   (se non esiste ritorna -1)
*/

private static int nextMatch (int score[], int n)
  {
    boolean ok = true;

    do
      {
       n++;
       if (n >= score.length)
         {
          ok =false;
          break;
```

```
        }
      }
    while (score[n] <= 0);

    if (!ok) return -1;
    else return n;
   }


/**
  Estrae il vettore di distanze dal vettore delle frasi
*/

private static Vector extractDist (Vector vLines, String delim)
 {
   Vector vDist = new Vector();

   for (int i=0; i<vLines.size(); i++)
      {
       String line = (String) vLines.elementAt(i);
       if (line.startsWith(delim))          // linea contenente informazioni di distanza
          {
           String num = line.substring (delim.length()+1);
           int dist = Integer.parseInt (num);
           vDist.addElement(new Integer(dist));
          }
      }
   return vDist;
 }


/**
  Restituisce il vettore delle frasi privato delle linee
  contenenti le informazioni sulla distanza
*/

private static Vector removeDist (Vector vLines, String delim)
 {
   for (int i=0; i<vLines.size(); i++)
      {
       String line = (String) vLines.elementAt(i);
       if (line.startsWith(delim))       // linea contenente informazioni di distanza
          {vLines.removeElementAt(i);
           i--;
          }
      }
   return (vLines);
 }


/**
  Restituisce il vettore delle frasi privato delle linee
  contenenti i commenti
```

```
*/

private static Vector removeComments (Vector vLines)
 {
   for (int i=0; i<vLines.size (); i++)
      {
       String line = (String) vLines.elementAt(i);
       if ( line .startsWith("##"))              // linea contenente commenti
          {vLines.removeElementAt(i);
           i--;
          }
      }
   return (vLines);
 }


/**
  Calcola la Longest Common Substring (in caratteri)
  tra due stringhe (parole)
*/

private static int lcs (String s, String t)
 {
   int n = s.length ();      // lunghezza di s
   int m = t.length ();      // lunghezza di t

   int d [][] = new int[n+1][m+1]; // matrice per il calcolo
   int i;      // itera in s
   int j;      // itera in t
   char s_i;  // iesimo carattere di s
   char t_j;  // jesimo carattere di t

   for (i = 0; i <= n; i++)    // inizializzazione prima colonna
         {d[i][0] = 0;}
   for (j = 0; j <= m; j++)    // inizializzazione prima riga
         {d[0][j] = 0; }

   for (i = 1; i <= n; i++)          // calcolo matrice
         {s_i = s.charAt (i - 1);
          for (j = 1; j <= m; j++)
                {t_j = t.charAt (j - 1);
                 if ( s_i == t_j) { d[i][j] = d[i-1][j-1] + 1; }
                 else { d[i][j] = Math.max(d[i-1][j],d[i][j-1]); }
                }
         }

   return d[n][m];
 }


 /**
   Estrae dalla regione (frase) i token, suddivisi in:
```

```
     * punteggiatura
     * stringhe con almeno un carattere numerico
     * le rimanenti parole di lunghezza almeno MINLEN caratteri

   Restituisce un vettore di token per ciascuna categoria
   e il numero totale di parole della regione
*/

private static Vector findTokens (String sent)
 {
  final int MINTOKLEN = Main.MainFrame.MINTOKLEN; // minima lunghezza di una parola per
                                                  // effettuare l'allineamento di
                                                  // somiglianza

  Vector vTokens = new Vector();
  Vector vPunct = new Vector();    // punteggiatura
  Vector vNum = new Vector();      // stringhe con almeno un carattere numerico
  Vector vWords = new Vector();    // rimanenti parole di lunghezza almeno MINLEN caratteri

  int tokCount = 0;

  StringTokenizer st = new StringTokenizer (sent);

  while (st.hasMoreTokens())
    {
     String tok = st.nextToken();

     if (punctCheck(tok)) vPunct.addElement (new Token(tokCount, tok));
     else if (numCheck(tok)) vNum.addElement (new Token(tokCount, tok));
     else if (tok.length() >= MINTOKLEN) vWords.addElement (new Token(tokCount, tok));

     tokCount++;
    }

  vTokens.addElement (vPunct);
  vTokens.addElement (vNum);
  vTokens.addElement (vWords);
  vTokens.addElement (new Integer(tokCount));

  return vTokens;
 }


/**
   Verifica se il token avuto in ingresso contiene
   almeno un carattere numerico
*/

private static boolean numCheck (String token)
 {
  boolean num = false;
  int cont=0;
```

```
    while ((!num) && (cont<token.length()))
     {
      if (Character.isDigit(token.charAt(cont))) num=true;
      cont=cont+1;
     }

    return num;
   }


  /**
    Verifica se il token avuto in ingresso un
    carattere di punteggiatura
  */

  private static boolean punctCheck (String token)
   {
    if (token.length() == 1 &&
          (token.equals(".") || token.equals(";") ||
           token.equals(":") || token.equals(",") ||
           token.equals("?") || token.equals("!") ||
           token.equals("(") || token.equals(")") ||
           token.equals("\""))
        ) return true;
    else return false;
   }


  /**
    Restituisce una stringa per ogni sottoregione trovata
    (il delimitatore quello specificato)
  */

  private static Vector mergeRegions(Vector vRegion, String delimiter)
   {
    String s = new String();
    Vector vResult = new Vector();

    for (int i=0; i<vRegion.size(); i++)
      {
       if (((String) vRegion.elementAt(i)).equals (delimiter)) // linea di delimitazione
         {
          vResult.addElement((String) s);
          s = new String();
         }
       else s = s + (s.equals("") ? "" : " ")                 // frase da appendere
                  + ((String) vRegion.elementAt(i));
      }
    return(vResult);
   }
}
```

A.2.5 Classe Align.WordAlignment

```
package Align;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                        WordAlignment
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
    Classe che implementa l'oggetto allinemento tra parole
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

public class WordAlignment
{
  private int pos[];
  private int score [];
  private int targetLen;


  /**
    Costruttore
  */

  WordAlignment (int n, int l)
   {
    pos = new int[n];
    score = new int[n];
    targetLen = l;

    for (int i=0; i<n; i++)
        {
         pos[i] = -1;
         score[i] = 0;
        }
   }

  WordAlignment (int p[], int s [], int l)
   {
    pos = p;
    score = s;
    targetLen = l;
   }

  /**
     Restituisce il vettore di posizioni
  */

  public int[] getPos ()
   {
     return pos;
   }

  /**
```

```
    Restituisce il vettore di punteggi
  */

  public int[] getScore ()
   {
     return score;
   }

  /**
    Restituisce la lunghezza della frase target
  */

  public int getTargetLen ()
   {
     return targetLen;
   }
}
```

A.3 Package GlobalUtility

A.3.1 Classe GlobalUtility.GlobalUtility

```
package GlobalUtility;

/**
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                        GlobalUtility
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
      Classe astratta contenente funzioni di utilita '
      per la lettura di file
    /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

import java.util.*;
import java.io.*;

public abstract class GlobalUtility
{
   /**
     Restituisce un vettore contenente le linee lette nel file
   */

   public static Vector readLines(FileReader File) throws IOException
    {
     BufferedReader in = new BufferedReader (File);

     String line = null;
     Vector vLines = new Vector();
```

```
  line = in.readLine();

  while (line != null)
    {
     vLines.addElement((String) line);
     line = in.readLine();
    }

  return vLines;
 }


/**
  Restituisce un vettore contenente le frasi lette nel file
  (vettore di oggetti TMPhrase)
*/

public static Vector readTMFile(FileReader File) throws IOException
 {
  BufferedReader in = new BufferedReader (File);

  String phraseOrig = new String();      // frase da tradurre
  String phraseTrad = new String();      // frase tradotta
  String aPos = new String();            // stringa di posizioni di allineamento
  String aScore = new String();          // stringa di punteggi di allineamento

  Vector vTMPhrases = new Vector();

  phraseOrig = in.readLine();

  while (phraseOrig != null)
    {
     phraseTrad = in.readLine();
     aPos = in.readLine();
     aScore = in.readLine();

     vTMPhrases.addElement(new TMPhrase(phraseOrig, phraseTrad, aPos, aScore));
     phraseOrig = in.readLine();
    }

  return vTMPhrases;
 }


/**
  Automa deterministico che separa le frasi in un documento di testo
  e restituisce un vettore con le frasi estratte
  La condizione di separazione tra una frase e l'altra :
  (. ? !) seguiti da spazio o invio, oppure doppio invio
  (quest'ultima condizione corrisponde anche alla fine del paragrafo)
*/

public static Vector readPhrases (FileReader FileR) throws IOException
```

```
{
 final int CR = 13;      // carriage return
 final int LF = 10;      // line feed

 int i;
 char c;
 String s;

 BufferedReader in = new BufferedReader(FileR);
 StringBuffer phrase = new StringBuffer();
 Vector vPhrases = new Vector();

 int stato = 0;          // stato iniziale

 while ((i = in.read()) != -1)        // lettura del file
    {
     c = (char) i;
     switch (stato)
         {case 0: if (c == CR)
                     {
                      in.read();      // scarta il successivo LF
                      stato = 3;
                     }
                  else if (c == '.' || c == '!' || c == '?')
                     {
                      phrase.append(c);
                      stato = 1;
                     }
                  else
                     {
                      phrase.append(c);
                     };
                  break;

          case 1: if (c == CR)
                     {
                      in.read();      // scarta il successivo LF
                      stato = 2;
                     }
                  else if (c == ' ')
                     {
                      stato = 2;
                     }
                  else
                     {
                      phrase.append(c);
                      stato = 0;
                     };
                  break;

          case 2: vPhrases.add(phrase.toString());     // aggiunta della frase
                  phrase = new StringBuffer();
```

```
            if (c == CR)
               {
                in.read();        // scarta il successivo LF
                stato = 4;
               }
            else if (c == ' ')
               {
                stato = 4;
               }
            else
               {
                phrase.append(c);
                stato = 0;
               };
            break;

case 3:     if (c == CR)
               {
                in.read();        // scarta il successivo LF
                stato = 5;
               }
            else
               {
                phrase.append(" "+c);
                stato = 0;
               };
            break;

case 4:     if (c == CR)
               {
                in.read();        // scarta il successivo LF
                stato = 4;
               }
            else if (c == ' ')
               {
                stato = 4;
               }
            else
               {
                phrase.append(c);
                stato = 0;
               };
            break;

case 5:     vPhrases.add(phrase.toString());     // aggiunta della frase
            vPhrases.add("");                    // fine paragrafo
            phrase = new StringBuffer();

            if (c == CR)
               {
                in.read();        // scarta il successivo LF
                stato = 4;
               }
```

```
                    else if (c == ' ')
                       {
                        stato = 4;
                       }
                    else
                       {
                        phrase.append(c);
                        stato = 0;
                       };
                    break;
            }
        };

    if (phrase.length() > 0)        // aggiunta frase finale (se presente)
                {
                 vPhrases.add(phrase.toString());
                }

    return vPhrases;
   }


/**
  Restituisce il vettore delle frasi con la punteggiatura di ciascuna
  messa in evidenza (ad es. "aaa, bbb, ccc." -> "aaa , bbb , ccc ."),
  per il corretto riconoscimento dei token.
*/

public static Vector separatePunct (Vector vSent)
 {
   for (int i=0; i<vSent.size(); i++)
      {
       String line = (String) vSent.elementAt(i);
       if (line.equals("")) continue;

       String mLine = new String();

       if (line.charAt(0) == '(') mLine = mLine + "( ";
       else mLine = mLine + String.valueOf (line.charAt(0));

                              // separazione della punteggiatura che si postpone
                              //  "."  ";"  ":"  ","  "?"  "!"  ")" """ "'"
                              // e della punteggiatura che si prepone
                              //  "(" """
       for (int j=1; j<line.length()-1; j++)
          {
           if (line.charAt(j) == '.' && line.charAt(j+1) == ' ')
              {mLine = mLine + " . "; j++;}
           else if (line.charAt(j) == ';' && line.charAt(j+1) == ' ')
              {mLine = mLine + " ; "; j++;}
           else if (line.charAt(j) == ':' && line.charAt(j+1) == ' ')
              {mLine = mLine + " : "; j++;}
           else if (line.charAt(j) == ',' && line.charAt(j+1) == ' ')
```

```
            {mLine = mLine + " , "; j++;}
          else if ( line.charAt(j) == '?' && line.charAt(j+1) == ' ')
            {mLine = mLine + " ? "; j++;}
          else if ( line.charAt(j) == '!' && line.charAt(j+1) == ' ')
            {mLine = mLine + " ! "; j++;}
          else if ( line.charAt(j) == ')')
            {mLine = mLine + " ) ";}
          else if ( line.charAt(j) == '"' || line.charAt(j) == ''
                || line.charAt(j) == '')
            {mLine = mLine + " \" ";}
          //else if (line.charAt(j) == '\")
          //   {mLine = mLine + "\' ";}
          else if ( line.charAt(j) == ' ' && line.charAt(j+1) == '(')
            {mLine = mLine + " ( "; j++;}
          else if ( line.charAt(j) == '(')
            {mLine = mLine + " ( ";}
          else mLine = mLine + String.valueOf (line.charAt(j));
         }

            // separazione punteggiatura terminale (non seguita da spazio)
        if ( line.charAt(line.length()-1) == '.' ||
             line.charAt(line.length()-1) == ';' ||
             line.charAt(line.length()-1) == ':' ||
             line.charAt(line.length()-1) == ',' ||
             line.charAt(line.length()-1) == '?' ||
             line.charAt(line.length()-1) == '!' ||
             line.charAt(line.length()-1) == ')' ||
             line.charAt(line.length()-1) == '"')
                mLine = mLine + " " + String.valueOf (line.charAt(line.length()-1));
        else if ( line.charAt(line.length()-1) == '' ||
                  line.charAt(line.length()-1) == '')
                  mLine = mLine + " \"";
        else mLine = mLine + String.valueOf (line.charAt(line.length()-1));

            // inserimento della frase elaborata in luogo di quella originale

        vSent.removeElementAt(i);
        vSent.insertElementAt((String) mLine, i);
       }
    return (vSent);
  }


/**
   Restituisce il vettore delle frasi con gli spazi multipli rimossi
*/

public static Vector cleanSpaces (Vector vSent)
 {
    for (int i=0; i<vSent.size(); i++)
       {
        String phr = (String) vSent.elementAt(i);
        String clphr = Stemming.eng.OperazSuFrase.filtraDoppiSpazi(phr);
```

```
            vSent.removeElementAt(i);
            vSent.insertElementAt((String) clphr , i );
           }
       return (vSent);
    }
}
```

A.3.2 Classe GlobalUtility.TMPhrase

```
package GlobalUtility;

/**
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
                              TMPhrase
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
     Classe che implementa l'oggetto "Translation
     Memory Phrase"
     /\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
*/

public class TMPhrase
{
  private String phraseOrig;
  private String phraseTrad;
  private String aPos;
  private String aScore;

  /**
    Costruttore
  */

  public TMPhrase (String po, String pt, String ap, String as)
   {
    phraseOrig = po;
    phraseTrad = pt;
    aPos = ap;
    aScore = as;
   }

  /**
     Restituisce la frase originaria (da tradurre)
  */

  public String getPhraseOrig ()
   {
     return phraseOrig;
   }

  /**
     Restituisce la frase tradotta
```

```
  */

  public String getPhraseTrad ()
   {
     return phraseTrad;
   }

  /**
     Restituisce la stringa delle posizioni di allineamento
  */

  public String getAPos ()
   {
     return aPos;
   }

  /**
     Restituisce la stringa dei punteggi di allineamento
  */

  public String getAScore ()
   {
     return aScore;
   }
}
```

Appendice B

Il codice SQL

B.1 Gli script

B.1.1 Creazione delle tabelle

```
CREATE TABLE <phrase1>          -- generica tabella delle frasi
  (codice NUMBER (10) PRIMARY KEY,
   fraseorig VARCHAR2 (1000),
   frase VARCHAR2 (1000),
   frasetrad VARCHAR2 (1000),
   wordlen VARCHAR2 (1000),
   apos VARCHAR2 (700),
   ascore VARCHAR2 (700),
   spos VARCHAR2 (700)
  )

CREATE TABLE <qphrase1>         -- generica tabella dei q-grammi
  (codice NUMBER(10) REFERENCES <phrase1> ON DELETE CASCADE,
   pos NUMBER (3),
   qgram VARCHAR2 (<size>),
   PRIMARY KEY (codice,pos)
  )

CREATE TABLE FULLMATCH
  (cod2 NUMBER(10),
   cod1 NUMBER(10),
   dist NUMBER(3),
   PRIMARY KEY (cod2, cod1),
   FOREIGN KEY (cod1) REFERENCES PHRASE1 (codice) ON DELETE CASCADE,
   FOREIGN KEY (cod2) REFERENCES PHRASE2 (codice) ON DELETE CASCADE
  )
```

```
CREATE TABLE MATCHPOS
  (cod1     NUMBER(10),
   cod2     NUMBER(10),
   nr       NUMBER(3),
   nc       NUMBER(3),
   PRIMARY KEY (cod1, cod2, nr, nc),
   FOREIGN KEY (cod1) REFERENCES PHRASE1 (codice) ON DELETE CASCADE,
   FOREIGN KEY (cod2) REFERENCES PHRASE2 (codice) ON DELETE CASCADE
  )

CREATE TABLE SUBMATCH
  (cod2     NUMBER(10),
   i2       NUMBER(3),
   f2       NUMBER(3),
   cod1     NUMBER(10),
   i1       NUMBER(3),
   f1       NUMBER(3),
   dist     NUMBER(3),
   PRIMARY KEY (cod2, i2, cod1, i1),
   FOREIGN KEY (cod1, i1, cod2, i2) REFERENCES
           MATCHPOS (cod1, nr, cod2, nc) ON DELETE CASCADE,
   FOREIGN KEY (cod1, f1, cod2, f2) REFERENCES
           MATCHPOS (cod1, nr, cod2, nc) ON DELETE CASCADE,
   FOREIGN KEY (cod2) REFERENCES PHRASE2 (codice) ON DELETE CASCADE
  )
```

B.2 Creazione delle stored procedure

```
CREATE OR REPLACE FUNCTION wordLen (frase VARCHAR2)
  RETURN NUMBER
AS LANGUAGE JAVA NAME 'SimSearch.DBUtility.wordLen
(java.lang.String)
  return int';

CREATE OR REPLACE FUNCTION wordSubString (frase VARCHAR2, da
NUMBER, a NUMBER)
  RETURN VARCHAR2
AS LANGUAGE JAVA NAME 'SimSearch.DBUtility.wordSubString
(java.lang.String, int, int)
  return java.lang.String';

CREATE OR REPLACE FUNCTION transPos (frase VARCHAR2, n NUMBER)
  RETURN NUMBER
AS LANGUAGE JAVA NAME 'SimSearch.DBUtility.transPos
(java.lang.String, int)
  return int';

CREATE OR REPLACE FUNCTION wordEditDistanceDiag (stringa1
VARCHAR2, stringa2 VARCHAR2, maxd NUMBER)
  RETURN NUMBER
```

```
AS LANGUAGE JAVA NAME 'SimSearch.Distance.wordEditDistanceDiag
(java.lang.String, java.lang.String, double)
  return int';

CREATE OR REPLACE FUNCTION wordEditDistanceSubCheck (s1 VARCHAR2,
c1 NUMBER, s2 VARCHAR2, c2 NUMBER, maxd NUMBER)
  RETURN NUMBER
AS LANGUAGE JAVA NAME 'SimSearch.Distance.wordEditDistanceSubCheck
(java.lang.String, int, java.lang.String, int, double)
  return int';
```

B.3 Le query

B.3.1 Ricerca full - primo passo

```
INSERT INTO FULLMATCH
SELECT r2.codice AS cod2, r1.codice AS cod1,
      wordEditDistanceDiag (r1.frase, r2.frase, <k>)
FROM <tab1> r1, <qtab1> r1q, <tab2> r2, <qtab2> r2q
WHERE r1.codice = r1q.codice
AND r2.codice = r2q.codice
AND r1q.qgram = r2q.qgram

                                        -- filtro di posizione
AND ABS (r1q.pos - r2q.pos) <= ROUND(<k> * r2.wordLen)

                                        -- filtro di lunghezza
AND ABS (r1.wordLen - r2.wordLen) <= ROUND(<k> * r2.wordLen)

GROUP BY r2.codice, r1.codice, r1.frase, r2.frase, r1.wordLen, r2.wordLen

                                        -- filtro di conteggio
HAVING COUNT(*) >=
         (r1.wordLen - 1 - (ROUND(<k> * r2.wordLen) - 1) * <q>)
AND COUNT(*) >=
         (r2.wordLen - 1 - (ROUND(<k> * r2.wordLen) - 1) * <q>)

-- (solo ricerca full)
AND wordEditDistanceDiag (r1.frase, r2.frase, <k>) >= 0

-- (ricerca full + preparazione sub)
AND wordEditDistanceSubCheck (r1.frase, r1.codice, r2.frase, r2.codice, <k>) = 0
```

B.3.2 Ricerca full - secondo passo

```
SELECT r2.codice AS cod2, r1.codice AS cod1,
      wordEditDistanceDiag (r1.frase, r2.frase, <k>)
FROM <tab1> r1, <tab2> r2, <qtab1> r1q, <qtab2> r2q
```

```
WHERE r1.codice = r1q.codice
AND r2.codice = r2q.codice
AND r1q.qgram = r2q.qgram
                                        -- filtro di ridondanza
AND r2.codice NOT IN (SELECT cod2 FROM FULLMATCH)

                                        -- filtro di conteggio
GROUP BY r2.codice, r1.codice, r1.frase, r2.frase
HAVING COUNT(*) >= <lMinSub> - 1 - (ROUND(<kSub>*<lMinSub>)-1)*<q> - (<q>-1)*2

AND wordEditDistanceSubCheck (r1.frase, r1.codice, r2.frase, r2.codice, <k>) >= 0
```

B.3.3 Ricerca sub

```
INSERT INTO SUBMATCH
  SELECT m1.cod2, m1.nc, m2.nc, m1.cod1, m1.nr, m2.nr,
         wordEditDistanceDiag (wordSubString(r1.frase, m1.nr, m2.nr),
         wordSubString(r2.frase, m1.nc, m2.nc), <kSub>)
  FROM MATCHPOS m1, MATCHPOS m2, <tab1> r1, <tab2> r2
  WHERE m1.cod1 = m2.cod1
  AND m1.cod2 = m2.cod2
  AND m1.cod1 = r1.codice
  AND m1.cod2 = r2.codice

                                        -- controllo estremi
  AND m1.nr < m2.nr
  AND m1.nc < m2.nc
                                        -- controllo lunghezza minima
  AND (m2.nc - m1.nc + 1) >= <lMinSub>
  AND (m2.nr - m1.nr + 1) >= <lMinSub>

                                        -- filtro di lunghezza
  AND ABS((m2.nc - m1.nc) - (m2.nr - m1.nr)) <= ROUND((m2.nc - m1.nc +1)*<kSub>)

                                        -- filtro di conteggio
  AND (SELECT COUNT(*)
       FROM MATCHPOS m1a
       WHERE m1a.cod1 = m1.cod1
       AND m1a.cod2 = m1.cod2
       AND m1a.nc <= m2.nc
       AND m1a.nc >= m1.nc
       AND m1a.nr <= m2.nr
       AND m1a.nr >= m1.nr
                                        -- filtro di posizione
       AND ABS((m1a.nc - m1.nc)-(m1a.nr - m1.nr)) <=ROUND((m2.nc - m1.nc +1)*<kSub>)
    ) >= ALL ((m2.nc - m1.nc +1) - ROUND((m2.nc - m1.nc +1)*<kSub>),
              (m2.nr - m1.nr +1) - ROUND((m2.nc - m1.nc +1)*<kSub>))
  AND wordEditDistanceDiag (wordSubString(r1.frase, m1.nr, m2.nr),
          wordSubString(r2.frase, m1.nc, m2.nc), <kSub>) >= 0
```

```
                                        -- filtro di inclusione
AND NOT EXISTS
    (SELECT m3.nc, m3.nr, m4.nc, m4.nr
     FROM MATCHPOS m3, MATCHPOS m4, <tab1> r3, <tab2> r4
     WHERE m3.cod1 = m4.cod1
     AND m3.cod2 = m4.cod2
     AND m3.cod1 = r3.codice
     AND m3.cod2 = r4.codice
     AND m3.nr < m4.nr
     AND m3.nc < m4.nc
     AND (m4.nr - m3.nr + 1) >= <lMinSub>
     AND ABS((m4.nc - m3.nc)-(m4.nr - m3.nr)) <=ROUND ((m4.nc - m3.nc +1)*<kSub>)
     AND (SELECT COUNT(*)
            FROM MATCHPOS m3a
            WHERE m3a.cod1 = m3.cod1
            AND m3a.cod2 = m3.cod2
            AND m3a.nc <= m4.nc
            AND m3a.nc >= m3.nc
            AND m3a.nr <= m4.nr
            AND m3a.nr >= m3.nr
            AND ABS((m3a.nc -m3.nc)-(m3a.nr -m3.nr)) <=ROUND(<kSub>*(m4.nc-m3.nc +1))
        ) >= ALL ((m4.nc - m3.nc +1) - ROUND((m4.nc - m3.nc +1)*<kSub>),
                  (m4.nr - m3.nr +1) - ROUND((m4.nc - m3.nc +1)*<kSub>))
     AND wordEditDistanceDiag (wordSubString(r3.frase, m3.nr, m4.nr),
              wordSubString(r4.frase, m3.nc, m4.nc), <kSub>) >= 0
     AND m3.cod2 = m1.cod2
     AND ((m1.nc = m3.nc AND m4.nc > m2.nc)
          OR
          (m3.nc < m1.nc AND m2.nc = m4.nc)
          OR
          (m3.nc < m1.nc AND m4.nc > m2.nc)
          OR
          (m1.nc = m3.nc AND m2.nc = m4.nc AND m1.cod1 = m3.cod1 AND
             ((m1.nr = m3.nr AND m4.nr > m2.nr)
              OR
              (m3.nr < m1.nr AND m2.nr = m4.nr)
              OR
              (m3.nr < m1.nr AND m4.nr > m2.nr)
             )
          )
         )
    )
```

B.3.4 Estrazione risultati

```
SELECT fm.cod2 AS codice2,              -- estrazione risultati frasi intere
       r2.frase AS frase2, '' AS sfrase2,
       r2.fraseorig AS fraseorig2, '' AS sfraseorig2,
    fm.cod1 AS codice1,
       r1.frase AS frase1, '' AS sfrase1,
       r1.fraseorig  AS fraseorig1, '' AS sfraseorig1,
       r1.frasetrad AS frasetrad1, '' AS sfrasetrad1,
```

```
        fm.dist  AS dist, (fm.dist  / r2.wordLen) AS distrel,
           0 AS inizio, 0  AS sub
FROM FULLMATCH fm, <tab1> r1, <tab2> r2
WHERE fm.cod1 = r1.codice
AND fm.cod2 = r2.codice
                                        -- eventuale estrazione risultati  sottoparti
UNION
SELECT sm.cod2 AS codice2,
           r2.frase  AS frase2,
           wordSubString(r2.frase, sm.i2, sm.f2) AS sfrase2,
           r2.fraseorig  AS fraseorig2,
           wordSubString(r2.fraseorig, transPos(r2.spos,sm.i2), transPos(r2.spos,sm.f2))
              AS sfraseorig2,
        sm.cod1 AS codice1,
           r1.frase  AS frase1,
           wordSubString(r1.frase, sm.i1, sm.f1) AS sfrase1,
           r1.fraseorig  AS fraseorig1,
           wordSubString(r1.fraseorig, transPos(r1.spos,sm.i1), transPos(r1.spos,sm.f1))
              AS sfraseorig1,
           r1.frasetrad  AS frasetrad1,
           wordSubString(r1.frasetrad,transPos(r1.apos,transPos(r1.spos,sm.i1)),
              transPos(r1.apos,transPos(r1.spos,sm.f1))) AS sfrasetrad1,
        sm.dist  AS dist, (sm.dist  / (sm.f2-sm.i2+1)) AS distrel,
           sm.i2 AS inizio, 1  AS sub
FROM SUBMATCH sm, <tab1> r1, <tab2> r2
WHERE sm.cod1 = r1.codice
AND sm.cod2 = r2.codice

ORDER BY codice2, sub, inizio, dist, codice1
```

Bibliografia

[1] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions*. US Government Printing Office, 1964.

[2] A.T. Arampatzis, T. Tsoris, C.H.A. Koster, and Th.P. van der Weide. Phrase-based Information Retrieval. In *Information Processing and Management*, 1998.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[4] C. Brew and D. McKelvie. Word-pair Extraction for Lexicography. In *NEMLAP*, 1996.

[5] R.D. Brown. Automated Dictionary Extraction for Knowledge-Free Example Based Translation. In *Proceedings of 7th International Conference on Theoretical and Methodological Issues in Machine Translation*, 1997.

[6] K.W. Church. Char align: A Program for Aligning Parallel Texts at the Character Level. In *Computational Linguistics*, 1993.

[7] R. Cole, J. Mariani, H. Uszkoreit, G. Varile, A. Zaenen, A. Zampolli, and V. Zue. *Survey of the State of the Art in Human Language Technology*. Cambridge University Press and Giardini, Online version: www.cse.org.edu/CSLU/HLTsurvey.html, 1997.

[8] W.B. Croft and D.D. Lewis. An Approach to Natural Language Processing for Document Retrieval. In *SIGIR*, 1987.

[9] Atril Deja Vu - Translation Memory and Productivity System. Home page http://www.atril.com.

[10] B.J. Dorr, P.W. Jordan, and J.W. Benoit. A Survey of Current Paradigms in Machine Translation. Technical report, University of Maryland, College Park, 1998.

[11] O. Furuse and H. Iida. Constituent Boundary Parsing for Example Based Machine Translation. In *Proceedings of 15th International Congress on Computational Linguistics*, 1994.

[12] W.A. Gale and K.W. Church. A Program for Aligning Sentences in Bilingual Corpora. In *Computational Linguistics*, 1993.

[13] 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.

[14] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *Proceedings of 27th VLDB Conference*, 2001.

[15] D.A. Grossman, O. Frieder, D.O. Holmes, and D.C. Roberts. Integrating Structured Data and Text: A Relational Approach. *Journal of the American Society of Information Science*, 1997.

[16] T.D. Hedden. Machine Translation: A Brief Introduction. Home page http://www.he.net/hedden.

[17] M. Janusz. Course on Natural Language Processing. Technical report, Cornell University, 2000.

[18] G. Kondrak. Alignment of Phonetic Sequences. Technical report, University of Toronto, 1999.

[19] K. Kukich. Techniques for Automatically Correcting Words in Text. In *ACM Computing Surveys*, 1992.

[20] S. Lawrence, K. Bollacher, and C. Lee Giles. Indexing and Retrieval of Scientific Literature. In *Proceedings of 8th International Conference on Information and Knowledge Management*, 1999.

[21] E. Macklovitch and G. Russel. What's been Forgotten in Translation Memory. In *AMTA*, 2000.

[22] F. Mandreoli, R. Martoglia, and P. Tiberio. Searching Similar (Sub) Sentences for Example-Based Machine Translation. Lavoro sottoposto al Convegno SEBD 2002.

[23] I.D. Melamed. Bitext Maps and Alignment via Pattern Recognition. In *Computational Linguistics*, 1999.

[24] G.A. Miller. WordNet: A Lexical Database for English. In *CACM 38*, 1995.

[25] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Five Papers on WordNet. Technical report, Princeton University's Cognitive Science Laboratory, 1993.

[26] M. Nagao. *A Framework of a Mechanical Translation between Japanese and English by Analogy Principle*. Nato Publications, 1984.

[27] G. Navarro. A guided Tour to Approximate String Matching. Technical report, Dept. of Computer Science, University of Chile, 2001.

[28] *Oracle 8i - SQL Reference*, 2000. Oracle Corporation.

[29] *Oracle 8i - JDBC Reference*, 2000. Oracle Corporation.

[30] *Oracle 8i - SQLJ Developer's Guide*, 1999. Oracle Corporation.

[31] *Oracle 8i - Java Developer's Guide*, 2000. Oracle Corporation.

[32] *Oracle 8i - Java Stored Procedures Developer's Guide*, 1999. Oracle Corporation.

[33] *Oracle 8i interMedia Text - Reference*, 1999. Oracle Corporation.

[34] C. Rick. A New Flexible Algorithm for the Longest Common Subsequence Problem. Technical report, University of Bonn, Computer Science Department IV, 1994.

[35] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[36] D. Sankoff. Matching Sequences under Deletion/Insertion Constraints. In *Proceedings of the National Academy of Sciences of the USA*, 1972.

[37] S. Sato. Example Based Translation of Technical Terms. In *Proceedings of 5th International Conference on Theoretical and Methodological Issues in Machine Translation*, 1993.

[38] M. Simard, G.F. Foster, and P. Isabelle. Using Cognates to Align Sentences in Bilingual Corpora. In *International Conference on Theoretical and Methodological Issues in Machine Translation*, 1992.

[39] O. Steele. JWordNet - An Object Oriented Java Interface to WordNet. Home page http://prdownloads.sourceforge.net/jwn, 1998.

[40] T. Strzalkowki and J.P. Carballo. Recent Developments in Natural Language Text Retrieval. In *TREC*, 1993.

[41] H. Sumita, O. Furuse, and H. Iida. An Example Based Disambiguation of Prepositional Phrase Attachment. In *Proceedings of 5th International Conference on Theoretical and Methodological Issues in Machine Translation*, 1993.

[42] E. Sutinen and J. Tarhio. On Using Q-gram Locations in Approximate String Matching. In *Proceedings of Third Annual European Symposium*, 1995.

[43] E. Sutinen and J. Tarhio. Filtration with Q-samples in Approximate String Matching. In *Combinatorial Pattern Matching, 7th annual symposium*, 1996.

[44] J. Tiedemann. Word Alignment Step by Step. Technical report, Uppsala University, Department of Linguistics, 1999.

[45] Trados Team Edition - Translation Memory Technologies. Home page http://www.trados.com.

[46] Cypresoft Trans Suite 2000. Home page http://www.cypresoft.com.

[47] E. Ukkonen. Algorithms for Approximate String Matching. In *Information and Control*, 1985.

[48] E. Ukkonen. Approximate String Matching with Q-grams and Maximal Matches. In *Theoretical Computer Science*, 1992.

[49] J. Ullman. A Binary n-gram Tecnique for Automatic Correction of Substitution, Deletion, Insertion and Reversal Errors in Words. *The Computer Journal*, 1977.

[50] S.K.M. Wong, W.Ziarko, and P.C.N. Wong. Generalized Vector Space Model in Information Retrieval. In *Proceedings of 8th ACM SIGIR Conference on Research and Development in Information Retrieval*, 1985.