Università degli studi
di Modena e Reggio Emilia

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tecniche di ranking per l'interrogazione approssimata di dati XML

Cristian Colli

Tesi di Laurea

Relatore:
Chiar.mo Prof. Paolo Tiberio

Controrelatore:
Chiar.mo Prof. Sonia Bergamaschi

Correlatore:
Dott. Federica Mandreoli
Ing. Riccardo Martoglia

Anno Accademico 2001/2002

Ai miei genitori,
Elio e Maria Rosa

Ringraziamenti

Ringrazio il Professor Paolo Tiberio e la Professoressa Sonia Bergamaschi per la disponibilità e la collaborazione fornita. Un grazie particolare alla Dottoressa Federica Mandreoli e all' Ingegnere Riccardo Martoglia per la costante assistenza.
Un bacio a Sandra per l'appoggio e la pazienza dimostrata nei miei confronti e un grosso abbraccio ai compagni di università Fausto, Fabiano, Domenico, Gabriele e Danilo coi quali ho diviso il cammino.

Parole chiave

Dati Semistrutturati
Tree Edit Distance
XML
Query approssimate
Ranking dei risultati

Indice

Introduzione **1**

I Introduzione al problema 3

1 XML **5**
1.1 Lo standard 5
1.2 DTD 7
1.3 Infoset 7
1.4 Parsing e validazione 8
1.5 Lo standard XPATH 8
1.6 I Linguaggi di stile: XSLT 9
1.7 Modellazione e normalizzazione dei documenti XML 11
1.8 I Linguaggi di interrogazione: XQUERY 12
1.9 Stato dell'arte 14
1.10 Query approssimate 16

2 Tree Matching **21**
2.1 Introduzione 21
2.2 Unordered tree inclusion problem 22
2.2.1 Unordered path inclusion problem 23
2.3 Tree Embedding 24
2.3.1 Tree Embedding Approssimato 25
2.4 Tree edit distance 29
2.5 Preliminari 29
2.6 Tree edit distance fra alberi unordered 32
2.6.1 Numerazione dei nodi di un albero 33
2.7 Mapping 34
2.8 Marking 37
2.9 Costo di un marking legale 39
2.10 Grafo Bipartito 40
2.11 Ricerca esaustiva del miglior marking 41

II Approccio alla risoluzione di query XML 45

3 Tree edit distance 47
3.1 Premesse . . . 47
3.1.1 Le finalità . . . 47
3.1.2 Rappresentazione degli alberi . . . 48
3.1.3 Forma a stringa rappresentativa di un albero . . . 52
3.1.4 Premesse all'algoritmo realizzato . . . 55
3.2 Marking . . . 56
3.2.1 Costruzione dei marking . . . 56
3.2.2 Algoritmo per la creazione dei marking . . . 58
3.3 Riduzione degli alberi . . . 59
3.3.1 Assunzioni per la riduzione dell'albero . . . 59
3.3.2 Risultati della riduzione dell'albero . . . 60
3.3.3 Algoritmo per la riduzione dell'albero . . . 61
3.4 Calcolo della Tree Edit Distance . . . 62
3.4.1 Calcolo della distanza tra due nodi . . . 62
3.4.2 Calcolo della distanza tra alberi ridotti . . . 64
3.5 Algoritmo ricorsivo per la ricerca del best mapping . . . 66
3.6 VLDC: variable length don't cares . . . 68

4 Realizzazione di un filtro basato sul contenuto 73
4.1 Limiti dell'algoritmo per il calcolo della tree edit distance . . . 73
4.2 Premesse alla ricerca basata sul contenuto . . . 78
4.2.1 Limiti alla dimensione della query . . . 80
4.3 Algoritmo per la ricerca basato sul contenuto . . . 81
4.4 Criteri di selezione del filtro . . . 86
4.5 Creazione del dizionario . . . 86
4.6 Realizzazione del dizionario . . . 88

5 Prove sperimentali e risultati ottenuti 89
5.1 Le collezioni usate nei test . . . 89
5.2 Risoluzione di una query approssimata . . . 91
5.2.1 Efficacia . . . 92
5.2.2 Efficienza . . . 96
5.2.3 Variazioni dei parametri . . . 102

Conclusioni e sviluppi futuri 105

Elenco delle figure

1.1 Flusso dei dati nella trasformazione 10
1.2 Esempio di interrogazione 15
1.3 Esempio di archivio dblp 17
1.4 Weighted tree patterns 19

2.1 Tree inclusion problem 23
2.2 Path inclusion problem 24
2.3 Esempio di interrogazione 28
2.4 Esempi di edit operations 31
2.5 Rapprsentazione di un albero mediante i suoi sottoalberi 34
2.6 Rappresentazione di un mapping 35
2.7 Riduzione di un albero 38
2.8 Grafo bipartito 41
2.9 Mapping tra alberi ridotti dopo BG 42

3.1 Esempio di albero: T1 50
3.2 Esempio di albero: T2 51
3.3 Esempio di albero 52
3.4 Esempio di marking non isomorfi 57
3.5 Esempio di marking isomorfi 58
3.6 Esempio di albero ridotto: RK(T2) 60
3.7 Identificazione dei sottoalberi tramite forma a stringa 66
3.8 Elemento confronto 68
3.9 Esempio di VLDC 69

4.1 Esempio di query 74
4.2 Ricerca delle sottoparti dell'albero dati T 76
4.3 Ricerca delle sottoparti dell'albero dati T 79
4.4 Ricerca delle sottoparti dell'albero dati T 80
4.5 Ricerca dei nodi dell'albero dati T simili alla radice AB 82

5.1 Albero dati T (*collezione interna*) 92
5.2 Albero query Q (*collezione interna*) 93
5.3 Esempio di query tree con 7 nodi (*collezione dblp*) 96

5.4 Esempio di data tree (*collezione dblp)* 97
5.5 Test di scalabilità: tempi di esecuzione 98
5.6 Test di scalabilità: occupazione memoria 99
5.7 Test di scalabilità: tempi di esecuzione 100
5.8 Test di scalabilità: occupazione memoria 101
5.9 Efficienza al variare del valore di soglia 102
5.10 Efficienza al variare del valore di parentela 103
5.11 Selettività del filtro al variare della soglia 104

Introduzione

XML è largamente riconosciuto come lo standard per lo scambio dati di domani. Questo, in particolare grazie alla sua capacità di rappresentare una grande varietà di sorgenti dati e da qui la sua utilizzazione come linguaggio in grado di integrare dati provenienti da sorgenti differenti. Le stringhe rimangono il tipo di dato prevalente nei documenti XML perciò le tradizionali inconsistenze tra gli attributi in stringhe, come gli errori di compilazione, sono sempre presenti. Allo stesso tempo però XML è in grado di esprimere la struttura di un documento.

È quindi possibile che due sorgenti XML diverse contengano gli stessi dati espressi con una struttura differente ed è importante essere in grado di riconoscere questa *similarità*. Inoltre anche due sorgenti XML che hanno la stessa DTD (Document Type Definition) possono non essere espresse dalla stessa struttura ad albero, questo perchè possono esservi elementi opzionali o attributi. Per riconoscere queste similarità, abbiamo bisogno di tecniche efficienti che eseguano il *matching approssimato* tra documenti XML, basate sulle caratteristiche strutturali degli elementi del documento XML. Abbiamo allora bisogno di tecniche che misurino tali similarità e le quantifichino determinando la distanza, o come viene ridefinita, la *dissimilianza* tra documenti.

Allo stesso modo è importante essere in grado di interrogare efficientemente i documenti XML selezionando solo quelli che risultano di interesse. Per alcune interrogazioni è necessario selezionare solo i risultati corretti, che soddifano tutti i requisiti dell'interrogazione. Per altre, potrebbero non esistere soluzioni totalmente corrette, e allora diventa necessario selezionare le risposte che lo sono solo in parte. Esiste quindi l'esigenza di introdurre un valore di *soglia* che limiti il numero di risposte da selezionare.

Diverse tecniche sono state proposte per eseguire interrogazioni su sorgenti di dati XML tuttavia, noi pensiamo che l'unica in grado di fornire risultati soddifacenti, sia quella che, considerando i documenti come alberi radicati ed

etichettati, ricorre al calcolo della *tree edit distance* per determinare il valore di dissimilianza tra la query e i documenti sui quali è eseguita. Pensiamo, che l'obiettivo della ricerca, debba essere quello di ritrovare risultati precisi e caratterizzati da un valore di dissimilianza che sia una *metrica*. Valore sul quale sia possibile e facile adottare filtri che permettano di limitare lo spazio di ricerca.

È necessario che a ognuno dei risultati ottenuti coincida un preciso valore di dissimilianza che permetta, infine, un correto *ranking* dei risultati che limiti le soluzione a tutte e sole quelle corrette e che rientrano nel valore di soglia fissato.

Per quanto riguarda la struttura della tesi, essa è organizzata come segue. Nella **Prima Parte** vengono introdotti gli argomenti di ricerca studiati per affrontare il progetto della tesi. In particolare, nel **Capitolo 1** viene descritto il linguaggio di markup XML come strumento in grado di rappresentare sia documenti testuali che strutturati, dando risalto ai linguaggi per la formulazione di query in esso presenti. Nel **Capitolo 2** vengono introdotti gli alberi quali strutture in grado di rappresentare qualsiasi sorgente strutturata, analizzando il problema del tree matching e lo stato dell'arte in questo campo con particolare riferimento agli algoritmi per il calcolo della tree edit distance.

Nella **Seconda Parte** della tesi viene poi descritto l'approccio seguito per realizzare un algoritmo in grado di calcolare una distanza fra gli alberi che sia una metrica (**Capitolo 3**). Viene poi sviluppato un filtro per limitare lo spazio di ricerca e migliorare l'efficienza senza rinunciare alla correttezza e completezza dei risultati (**Capitolo 4**).

Infine, nel **Capitolo 5** vengono descritti i test e le collezioni progettate per verificare le funzionalità del software, oltre ovviamente ai risultati conseguiti.

Parte I

Introduzione al problema

Capitolo 1

XML

1.1 Lo standard

XML (extensible markup language) è uno standard abbastanza recente, ideato da Tim Bray e Jean Pauli, pubblicato ufficialmente nel febbraio 1998 dal World Wide Web Council (W3C), il consorzio preposto alla definizione degli standard del World Wide Web. XML, inizialmente proposto come standard per la rappresentazione dei documneti testuali, si sta ora imponendo come infrastruttura generale del World Wide Web, per l'interscambio di informazioni tra le diverse applicazioni. XML stabilisce un insieme di convenzioni per definire diversi linguaggi di codifica, in base alle diverse tipologie di informazione da rappresentare. Associando ai dati di partenza informazioni descrittive (dette tag o marcatori) pone le premesse per la costruzione di inferenze automatiche sulla semantica dei dati (Semantic Web).

XML è un linguaggio dichiarativo, che definisce le varie parti del testo e le loro rispettive relazioni, più precisamnete è un linguaggio di markup mediante il quale è possibile associare a ciascuna parte del testo un marcatore (tag), che lo qualifica come un determinato elemento logico. Un documento XML ha quindi una struttura gerarchica nella quale ciascun elemento, delimitato da un tag di apertura e di chiusura, è contenuto in altri elementi, ed un elemento radice contiene tutti gli altri. Nell'esempio riportato in seguito l'elemento `<curriculum>` `</curriculum>` contiene tutti gli altri.

Esempio 1

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE curriculum SYSTEM "curriculum.dtd">
<curriculum>
   <dati_personali>
       <nome>Carlo</Nome>
       <cognome>Bianchi</cognome>
       <indirizzo>
           <loc tipo="via">Moscova</loc>
           <civico>40</civico>
           <cap>20121</cap>
           <comune>Milano</comune>
           <provincia>MI</provincia>
       </indirizzo>
       <email>carlo.bianchi@mail.it</email>
       <tel>02-3456789</tel>
       <cellulare>349-286652</cellulare>
       </fax>
       <nascita>
           <data>24-5-1965</data>
           <luogo>
               <comune>Milano<comune>
               <provincia>MI</provincia>
           </luogo>
       </nascita>
       <nazionalita>italiana<nazionalita>
       <stato_civile>coniugato<stato_civile>
   </dati_personali>
</curriculum>
```

I marcatori XML ricordano da vicino le *glosse* con le quali i copisti medievali commentavano e integravano i manoscritti. Nel contesto attuale sono *etichette informatiche* utili per classificare le porzioni del testo alle quali si riferiscono e assegnare loro un ben preciso significato, che può diventare oggetto di elaborazioni da parte di un computer.

La modalità di rappresentazione grafica di ciascun elemento logico del testo, associato ad una coppia di marcatori, è definita a parte da appositi linguaggi, detti fogli di stile. Diventa così possibile utilizzare uno stesso documento codificato in XML per diverse modalità di pubblicazione (carta, CD-ROM, WWW), semplicemente cambiando il foglio di stile associato. Il linguaggio di stile più usato e più sofisticato è XSL (*eXstensible Stylesheet Language*) che tratteremo più dettagliatamente in seguito.

In questa capacità di rappresentare la struttura logica di un documento risiede una delle ragioni della sua popolarità.

1.2 DTD

XML non prescrive i nomi dei diversi marcatori, ma solo la sintassi generica per la loro definizione ed il loro utilizzo nell'identificazione degli elementi del testo. È tuttavia possibile definire la peculiare struttura logica che identifica e descrive una topologia di documenti nella cosiddetta DTD (Document Type Definition), un insieme di specifiche che, situate all'inizio del documento codificato o in un documento a parte, stabiliscono quali sono i nomi ammissibili per i marcatori, quali i nomi dei loro attributi e quali relazioni di inclusione possono sussistere tra di loro.

Un documento che rispetta la sintassi generica di XML, senza riferirsi ad una specifica DTD, è detto *ben formato*. Un documento XML congruente rispetto ad una determinata DTD è detto *valido*.

1.3 Infoset

Oggi la parte considerata più importante dell'intero standard XML è il cosiddetto Infoset, che descrive i documneti XML come insiemi di oggetti astratti, che hanno una o più proprietà dotate di nomi convenzionali, senza fare alcuna ipotesi sul loro formato di memorizzazione a basso livello (detto anche formato di serializzazione) che può essere o meno quello canonico basato sui caratteri. L'intero documento XML costituisce un oggetto-documento paragonabile alla radice di un albero *multisorte*, cioè composto da oggetti di vario tipo: gli elementi XML, il loro contenuto, le istruzioni di elaborazione ed i commenti. Gli

oggetti-contenuto (assieme ai commenti e le istruzioni di elaborazione) costituiscono i nodi terminali (le foglie) dell'albero infoset, mentre gli oggetti-elementi fungono da nodi intermedi. La parte centrale di infoset, detta nucleo o XML information set core, elenca i blocchi e le proprietà fondamentali che devono essere riconosciute e gestite da tutte le applicazioni di XML.

1.4 Parsing e validazione

Il risultato della validazione di un documento XML rispetto una DTD o ad uno Schema (operazione di *parsing*) è una rappresentazione dell'infoset del documento in un formato più adatto all'elaborazione dei programmi informatici, rispetto al formato di serializzazione. Le due rappresentazioni più comuni di XML sono il Document Object Model (DOM) Level 2 e la Simple API (Application Program(ming) Interface) for XML (SAX). Entrambe queste tecniche si basano sulla traduzione delle astrazioni di infoset in un modello ad oggetti che evita ai programmatori di dover manipolare direttamente i caratteri del formato di serializzazione di XML. Sia SAX che DOM definiscono un insieme di interfaccie che permettono ai programmi di accedere all'insieme delle informazioni XML, ma differiscono in alcuni aspetti fondamentali. SAX traduce l'albero Infoset in un documento XML in una sequenza lineare di eventi. Un parser DOM traduce l'Infoset del file in un albero di oggetti. DOM è particolarmente adatto alle applicazioni che devono rappresentare interamente in memoria un documento XML, mentre le applicazioni basate su SAX non hanno bisogno di tenere in memoria l'intera rappresentazione dell' Infoset. Una seconda non trascurabile differenza deriva dal fatto che DOM è una raccomandazione del W3C e ha dietro di se il peso istituzionale di questa organizzazione, mentre SAX è uno standard di fatto.

1.5 Lo standard XPATH

La struttura gerarchica di Infoset può essere usata per individuare sottoinsiemi dei nodi di un documento attraverso un semplice linguaggio che descrive l'attraversamneto dell'albero. In questo modo, invece di scrivere le routine di estrazione e di attraversamento in un linguaggio di programmazione, gli sviluppatori possono ricorrere a semplici espressioni e lasciare fare tutto al parser XML.

Il linguaggio proposto dal W3C per estrarre sottoinsiemi di nodi dell'albero dei documenti XML è chiamato XPATH (XML Path Language 1.0).

Il costrutto più importante usato nelle espressioni XPATH è chiamato *percorso di posizionamento*. Proprio come i percorsi dei file system, i percorsi di posizionamento XPATH possono essere assoluti o relativi. Quelli assoluti iniziano con una barra obliqua `(/)` e indicano che la navigazione deve incominciare dal nodo radice nel modello ad albero. Un percorso di posizionamento relativo non inizia con una barra obliqua: ciò significa che l'attraversamento dell'albero deve iniziare dal nodo con cui inizia il percorso di posizionamento.

Rifancendoci all'esempio precedente di documento XML, andiamo a scrivere l'espressione XPATH che individua tutti gli elementi `<nome>` che sono figli di elementi `<dati_personali>` i quali, a loro volta, discendono dal nodo radice `<curriculum>`.

```
/curriculum/dati_personali/nome
```

Nel caso di questo semplice esempio, non sarebbe difficile scrivere un programma che esegue lo stesso compito usando le API SAX o DOM, ma nel caso di espressioni più complicate che includano *caratteri jolly* e *condizioni logiche*, aumentano i vantaggi di delegare il codice di attraversamento al parser XML. L'espressione che segue estrae i nodi Infoset `<localita>` discendenti dalla radice `<curriculum>` e dotati di un attributo tipo il cui contenuto sia `piazza`

```
/curriculum/*/localita/[@tipo='piazza']
```

Oltre ad essere utilizzato direttamente XPath viene presupposto dallo standard XSLT 1.0 per la trasformazione di documenti XML e dal nuovo linguaggio di interrogazione XQuery proposto dal W3C. XSLT usa XPath per identificare i sottoinsiemi di nodi di un documento di origine che saranno tradotti in parti del documento di output.

1.6 I Linguaggi di stile: XSLT

Come già detto XML è un linguaggio di markup mediante il quale è possibile associare a ciascuna parte di un testo, un marcatore (tag), che la qualifica com

un determinato elemento logico, senza preoccuparsi di come esso apparirà fisicamente nel documento. La modalità di rappresentazione grafica di ciascun elemento logico del testo, associato ad una coppia di marcatori, è definita a parte da appositi linguaggi, detti fogli di stile. Diventa così possibile utilizzare uno stesso documento codificato in XML per diverse modalità di pubblicazione, semplicemente cambiando il foglio di stile associato. Il linguaggio di stile più usato e più sofisticato è XSL (eXtensible Stylesheet Language), un vero e proprio linguaggio di programazione basato su regole di trasformazione, che consente non solo di decidere il formato grafico, ma anche di scegliere quali elementi visualizzare e in che ordine.

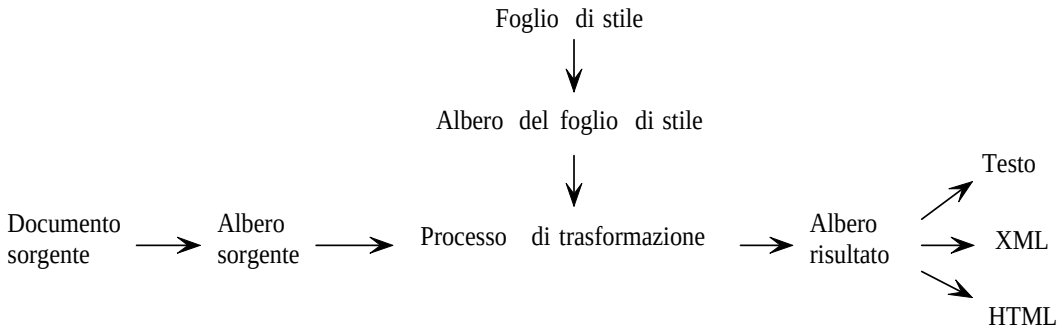


Figura 1.1: Flusso dei dati nella trasformazione

Nella definizione di questo standard divenne chiaro che si trattava di un processo in due fasi: nella prima la trasformazione strutturale del documento, nella seconda il processo di formattazione vero e proprio. Per questo il linguaggio XSL fu diviso in XSL-T per definire le trasformazioni e XSL-FO (XSL Formatting Object) per definire la fase di formattazione. La definizione di XSL-FO è però un'impresa di ragguardevole difficoltà e i prodotti che implementano XSL-FO sono in una fase di sviluppo ancora primitiva. Per questo i documenti XML vengono di solito tradotti usando XSL-T per produrre output ancora in HTML.

XSLT è un linguaggio standard progettato per trasformare documenti XML. L'elaborazione di un foglio di stile XSLT consiste di due fasi distinte: la trasformazione strutturale nella quale i dati XML in ingresso sono convertiti in una struttura che riflette l'output desiderato; la formattazione nella quale la nuova struttura è convertita nel formato di uscita richiesto, per esempio HTML o PDF.

La trasformazione strutturale consiste in operazioni come la selezione dei dati (cioè la scelta di alcuni degli elementi e attributi nel documento XML in ingresso), il loro ordinamento e l'esecuzione di coversioni aritmetiche.

XSLT presenta alcune somiglianze con i linguaggi di interrogazione usati per i database relazionali. In un database relazionale, i dati consistono in una serie di tabelle ed usando il linguaggio di interrogazione SQL (Structured Query Launguage), è facile definire operazioni di estrazione e di aggregazione dei dati delle tabelle per poi convertire il risultato in HTML. Come si verdrà in seguito, molti database relazionali oggi consentono di estrarre informazioni anche in formato XML. Il vantaggio di delegare a XSLT i compiti della trasformazione risiede nel fatto che i documenti trasformati sono adatti anche a sorgenti di dati eterogenee, perchè funzionano con qualunque sorgente di dati che esporti XML e non solo per i database relazionali.

Naturalmente XSLT e SQL sono destinati a coesistere: i dati saranno memorizzati in un database relazionale e trasmessi tra i sistemi in formato XML, personalizzandoli a seconda delle esigenze del ricevente.

1.7 Modellazione e normalizzazione dei documenti XML

Tutte le proposte per i nuovi linguaggi di interrogazione ricorrono ad una rappresentazione ad albero sia per la query che l'utente intende effetture sia per il documento XML su cui la query è da eseguire. È quindi necessasrio adottare un modello per la normalizzazione dei documenti XML in grado di modellare il documento e convertirlo in un albero radicato.

Il modello da noi adottato si può ricondurre a tutti quelli proposti in letteratura e in particolare a quello proposto da T. Shlieder in [11].

Il nostro modello considera i documenti XML formati da due tipi di nodi, nodi di testo (text node) che rappresentano gli elementi testuali come il valore degli attributi e nodi di struttura (struct node) che rappresentano gli elementi e gli attributi del documento XML.

L'attributo e in particolare il nome dell'attributo e degli elementi (struct node) sono inseriti come label dei nodi interni all'albero che si sta costruendo. I valori degli attributi (text node) sono inseriti come label di nodi che risulteranno le foglie dell'albero rappresentativo del documento.

Gli attributi del documento XML vengono quindi rappresentati nell'albero in due nodi tra loro in relazione padre-figlio. Il nome dell'attributo rappresenterà

la label del nodo padre mentre il valore dell'attributo formerà la label del nodo figlio.

1.8 I Linguaggi di interrogazione: XQUERY

Il problema di maneggiare ed interrogare efficientemente i documenti XML è tuttora una sfida aperta sia per i ricercatori che operano nell'information retrival sia per la comunità dei ricercatori operanti nell'ambito database. Una delle ragioni è che XML garantisce uno standard nella rappresentazione e nello scambio sia di documenti nel senso inteso dall'information retrival, sia nei dati prelevati da un database.

Esistono due contrapposte metodologie per la formulazione di un documento XML. Una detta *text-centric*, nella quale i documenti sono scritti con solo alcuni markup. Nell'altra, detta *data-centric*, i documenti possiedono una ben precisa struttura e generalmente vengono memorizzati in collezioni di documenti omogenee. XML è però anche considerato come il formato comune per l'integrazione tra una grande vastità di dati. Un particolare esempio sono i data warehouse che memorizzano in formato XML tutti i documenti, i messaggi o reports del database. Tutte queste collezioni di documenti sono data-centric ma non hanno uno schema comune e risultano quindi eterogenee.

Per ricercare documenti XML in collezioni di tipo text-centric, le tecniche adottate dall'information retrieval sono appropriate [2]. Per interrogare documenti XML data-centric in collezioni omogene sono stati sviluppati nuovi linguaggi [6] [3]. Comunque, per interrogare collezioni eterogenee di dati XML di tipo data-centric, ne le tecniche dell'information retrieval, ne i linguaggi per le interrogazioni attualmente sviluppati, sono adatti.

Il modello classico vettoriale dell'information retrieval ignora completamente la struttura che il documento ha. I linguaggi di interrogazione, d'altra parte, non sono in grado di riconoscere i documenti che solo parzialmente rispondono alla query. Il loro confronto delle strutture opera una decisione binaria distinguendo i documenti che hanno stessa struttura da quelli che la hanno diversa. Non è infatti presente un classificazione dei risultati basata sulla differenza che esiste fra il documento in esame e la query da risolvere. Alcuni risultati alla query possono di conseguenza essere tralasciati e quindi persi.

In generale, una interrogazione XML dovrebbe restituire il migliore risultato possibile. Qualora non esista un documento che soddisfi appieno i requisiti espressi nella query dovrebbe restituire quelli più simili in accordo con i criteri di similarità scelti. Il problema di ricerca di similarità tra parole chiave della query nei documenti testuali è stato analizzato per anni nell'ambito dell'information retrieval. Sfortunatamente la maggiorparte delle soluzioni proposte (con alcune recenti eccezioni) considera dati non strutturati (text centric) perdendo perciò la possibilità di permenttere una ricerca più precisa. Le tecniche dell'information retrieval ignorano completamente la struttura del documento e risultano quindi inappropriate per interrogazioni XML nelle quali è presente una struttura logica (data centric documents) [2].

Ci sono state quindi diverse proposte recenti per i linguaggi di interrogazione su documenti XML, ma finora nessuna è stata adottata come standard. Tuttavia esiste un lavoro recentemente proposto da World Wide Web Consortium (W3C), chiamato XQuery che potrebbe diventare lo standard nel prossimo futuro.

Alcuni delle possibili caratteristiche di una XQuery sono illustrate nell'esempio qui sotto riportato e preso dalle pagine web del W3C [5].

```
<bib>{
    for $b in document("http://www.bn.com")/bib/book
  where $b/publisher="Addison-Wesley"
    and $b/@year>1991
 return <book year={ $b/@year }> { $b//title } </book>
}</bib>
```

Questa XQuery ricerca tutti i documenti `book` pubblicati da `Addison-Wesley` dopo il `1991`. Per tutti quelli riconosciuti restituisce l'anno di pubblicazione ed il titolo.

L'espressione `document(URL)` ricerca il punto di accesso ai dati XML dell'intero documento XML che si trova localizzato all'indirizzo URL specificato. XQuery, come già detto ricorre alle espressioni di XPath per navigare attraverso il documento XML. Per esempio, `$b/publiscer` ricerca tutti i figli di `$b` con identificatore del tag `publisher`, mentre la selezione tramite *wildcard* `$b//title` ricerca tutti i discendenti di `$b` per i quali l'identificatore del tag è `title`.

1.9 Stato dell'arte

La grande diffusione dello standard di fatto XML ha portato alla necessità di sviluppare nuove tecniche efficienti per l'immagazzinamento e l'interrogazione di documenti XML. I ricercatori hanno proposto di usare i sistemi con database relazionali per soddifare a queste richieste attraverso la decomposizione dei documenti in formato XML. Lo scopo è quello di produrre dati in formato relazionale e trasformare le query XML in query SQL.

Ci sono alcuni prodotti commerciali che, avvantaggiandosi della già sviluppata tecnologia dei database management, sono in grado di memoririzzare e ricercare dati in formato XML, sebbene nessuno di questi supporta pienamente XQuery. In effetti oggi molti produttori di database relazionali forniscono già diverse funzionalità per memorizzare e maneggiare documenti in formato XML nei loro sistemi. Il più di questi sistemi supportano inserimenti automatici di dati XML strutturati in forma canonica direttamnete nelle tabelle. Forniscono inoltre metodi per esportare i dati nelle tabelle in formato XML e interrogare e trasformare in HTML tale formato.

Diversamente dai database (DB) convenzionali, i dati XML possono essere memorizzati in varie forme, sia nella loro forma nativa (come documento testuale), sia in una forma semi-strutturata secondo uno schema standard conforme al database, o in una specifica applicazione del database che esporta i suoi dati in formato XML per essere distribuiti dal server ai clienti.

Basandoci su precedenti esperienze possiamo affermare che con i database tradizionali, le interrogazioni possono essere ottimizzate più efficientemente se vengono prima trasformate in una forma interna adatta e con una semantica chiara.

Se i documenti XML sono memorizzati secondo uno schema da una applicazione specifica del database, le query XML su questi dati possono essere trasformate nel linguaggio nativo del sistema per formulare query.

Dall'altra parte, se i documenti XML sono memorizzati in forma semistrutturata o processati on the fly come stream di dati XML, nessuna delle algebre esistenti è adatta ed è necessario un nuovo tipo di algebra che sia in grado di riconoscere e interpretare le eterogeneità e le irregolarità intrinseche nella struttura dei dati XML.

Un documento XML può avere una struttura interna complessa e può essere visto come un albero radicato in cui esistono radice, nodi interni e foglie. I nodi dell'albero corrispondono agli elementi (attributi) mentre le diramazioni rappresentano le relazioni logiche del documento XML. Questa visione ad albero per i documenti XML risulta evidente nei linguaggi di interrogazione dell'XML stesso come Quilt, XQuery e XPath.

Nei linguaggi per interrogazioni su documenti XML anche le query da risolvere, vengono trattate come alberi radicati e il problema di eseguire una interrogazione su un documento XML viene quindi ricondotto alla ricerca dell'albero query nell'albero dati.

L'approccio al problema di risoluzione di una query può allora essere ricondotto alla ricerca delle sottoparti dell'albero dati simili all'albero query. Questo modo di procedere è seguito anche da Shasha in [14].

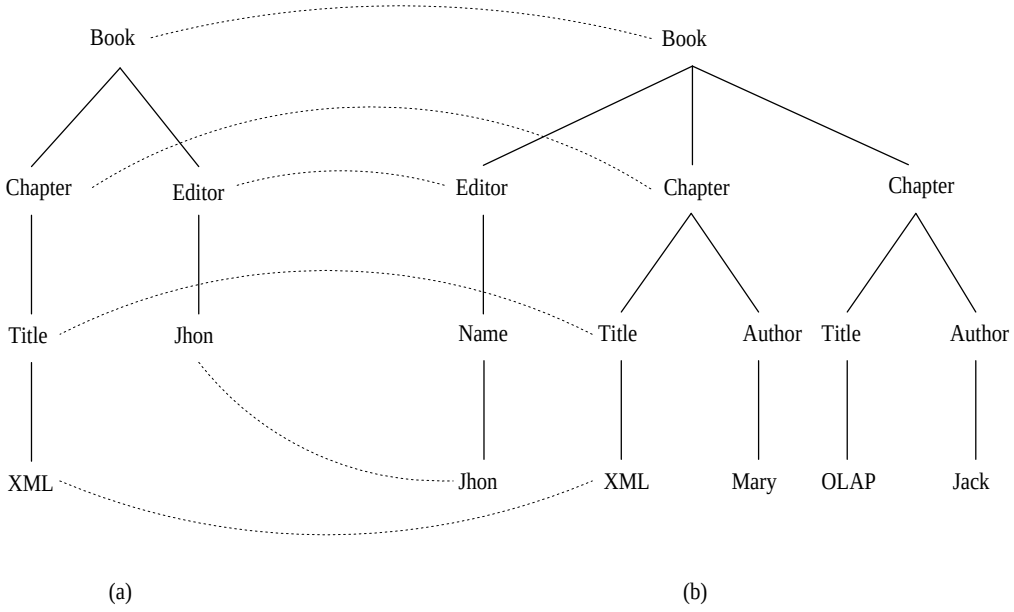


Figura 1.2: Esempio di interrogazione

Questo esempio di interrogazione può essere ricondotto all'*unordered tree inclusion problem* e che analizzeremo meglio al capitolo 2. Comunque si capisce facilmente che l'intento è quello di riconoscere le parti dell'albero dei dati XML `(b)` che rappresentano meglio l'albero query `(a)` e quindi risolvono la query.

Tutti i linguaggi di interrogazione hanno come obiettivo però quello di selezionare i documenti ritrovati nella ricerca sia secondo la loro struttura che secondo il valore dei loro elementi.

Esistono diverse proposte di linguaggi per l'esecuzioni di query XML approssimate che possono essere classificate in due tipi di approcci. Uno basato sul *contenuto* l'altro sulla *struttura gerarchica*. Nel primo viene data particolare importanza al *contenuto* del documento e quindi al valore dei suoi elementi. Si concentrano nel determinare quante ugualianze esistono tra gli elementi e calcolano la similarità del documento con la query basandosi sul numero e sulla qualità delle similarità. Il secondo approccio, ritiene importante la struttura che lega gli elementi del documento e ricerca la similarità con la query cercando quelle parti del documento che possiedono, prima di tutto, una *struttura gerarchica* analoga a quella della query.

Osserviamo però che mentre la struttura ad albero dei documenti XML è ordinata, le interrogazioni dovrebbero essere rappresentate tramite alberi non ordinati. In effetti, il più delle volte, l'ordine dei dati non è importante per l'utente che effettua la query. Questo è uno dei motivi per cui XQuery supporta la possibilità di risolvere query per alberi in cui l'ordine tra i fratelli non è importante.

1.10 Query approssimate

Iniziamo col considerare un primo approccio proposto dai ricercatori per la risoluzione di query approssimate su documenti XML in [1]. Come esempio concreto possiamo considerare di volere interrogare un database bibliografico, quale può essere DBLP [15]. Un utente protrebbe voler cercare tutti i `book` che hanno come sottoelemento un `isbn`, un `url` ed una `ee` (edizione elettronica). Alcuni di questi sottoelementi non devono obbligatoriamente comparire nella struttura di un elemento `book` e solo pochi di essi conterranno, al loro interno tutti gli elementi che possono contenere.

Inoltre, l'utente vorrebbe che le soluzioni alla query, che solo approssimativamente rispondono alla query, fossero classificate sencondo il grado di similarità con la query stessa.

I ricercatori, in questa proposta da noi considerata, credono che l'approccio migliore per risolvere una query approssimata sia quello di ricercare tutti i match approssimati del pattern della query e restituire una lista avendo selezionato le soluzioni trovate in base alla relativa distanza con il pattern della query.

In [1] viene rappresentata, come esempio, una porzione dell' archivio dblp e di seguito (figura: 1.3) anche noi la riportiamo.

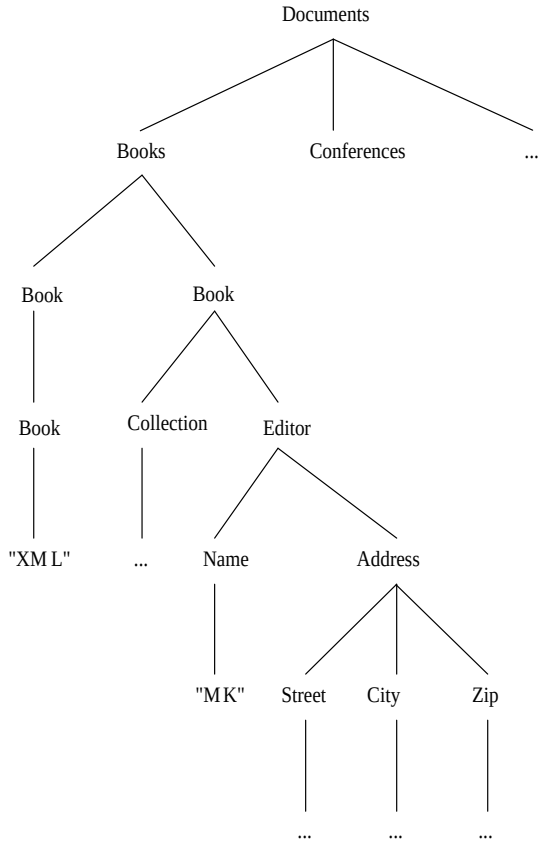


Figura 1.3: Esempio di archivio dblp

La tecnica proposta, per risolvere query non esatte ma approssimate, fonda i suoi presupposti su operazioni chiamate *tree-pattern-relaxation*. Queste operazioni cercano di rilassare il contenuto e la struttura della query stessa. Nel primo caso, ricercano un rilassamento del contenuto espresso dalla query, si potrebbe cercare di risolvere una query in cui compare *document* invece di *book*, nel secondo caso, volendo rilassare i vincoli espressi nella query, si potrebbe voler rilassare il vincolo padre figlio tra due elementi e permettere che gli stessi elementi siano invece in relazione di antenato discendente.

Entrambe le tipologie di operazioni sono volte a produrre query simili all'originaria, sulla quale però sono state eseguite operazioni di trasformazione al fine di rilassare la query e di produrre quindi *query rilassate*.

Una volta create le query rilassate è possibile, risolvendo in modo esatto queste ultime, risolvere la query iniziale in modo approssimato. Possiamo quindi affermare che si perviene al risultato attraverso una metodologia indiretta.

È quindi possibile classificare le risposte approssimate in base al numero di **tree-pattern relaxation** applicate alla corrispondente query rilassata. Siccome quantificare le operazioni di rilassamento per esprimere l'approssimazione tra la query originaria e quella approssimata non garantisce una buona metrica, per migliorare tale valutazione , vengono introdotti i **weighted tree patterns** aggiungendo flessibilità al processo di classificazione.

Associando a ciscun nodo ed edge del tree pattern della query un punteggio per il match esatto ed uno per quello approssimato, si permette di gestire un grado molto fine di controllo sul punteggio associato a ciascuna risposta approssimata alla query.

Associando a ciascuna query rilassata un punteggio, qualora esista un match esatto tra una relaxed query e l'albero dati, il punteggio della soluzione trovata nel match è uguale a quello della query rilassata.

Come detto il problema di risoluzione ad una query approssimata richiede di selezinare solo le risposte che rientrano nella soglia fissata prima dell'esecuzione. In questo caso, i ricercatori calcolano il punteggio di ciascuna risposta trovata e solo quelle con un punteggio *superiore* alla soglia vengono riportate come risposte alla query. Hanno quindi definito il *Threshold problem*

Definizione 1.1 ***Threshold problem:*** *Dato un query tree pattern Q pesato (in cui a ciascun nodo ed edge sono assegnati i relativi costi) e una soglia t il* threshold problem *consiste nel determinare tutte le risposte approssimate alla query Q con un* punteggio *superiore a t.*

Nell'esempio di *weighted tree pattern* riportato in figura 1.4, preso da [1], possiamo osservare che al nodo `book` sono assegnati due punteggi: (7,1). Nella query originale il punteggio vale 7 mentre in tutte le query rilassata in cui il valore del nodo `book` viene cambiato (con un sinonimo) il punteggio vale 1.

Considerando invece l'edge tra `book` ed `editor` possiamo osservare che per le query in cui la relazione non viene rilassata il punteggio vale 4 mentre se viene rilassata 3.

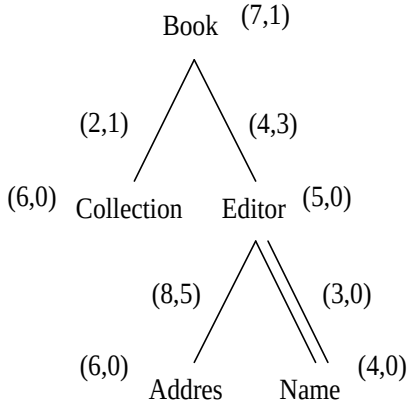


Figura 1.4: Weighted tree patterns

Possiamo ora osservare che mentre è sostanzialmente corretto il metodo di assegnazione dei punteggi per i rilassamenti sulla struttura, risultano poco efficaci i punteggi assegnati ai rilassamenti sul contenuto. Infatti la prima critica che si può muovere contro il modello proposto è quella verso l'assegnazione dello stesso punteggio a qualunque rilassamneto sul contenuto di un nodo senza differenziare la similarità che esiste tra il nodo della query originale ed i sinonimi che vengono introdotti nelle query rilassate. Inoltre non è chiaro chi deve decidere quali sono i sinonimi da sostituire nelle query rilassate e secondo quali metodologie.

Occorre poi precisare che una query o tree pattern potrebbe avere un grande numero di risposte approssimate, e non è quindi desiderabile, ricevere tutte le risposte approssimate alla query.

Quindi un ulteriore problema di questa proposta è il come determinare , dato un weighted query tree pattern, tutte le possibili query rilassate efficentemente e in modo da garantire che solo i risultati rilevanti siano restituiti. Non è infatti banale condiderare e costruire tutte le query rilassate. Le operazioni di rilassamento possono essere numerose e portare alla necessità di risolvere un elevato numero di query in modo esatto per trovare la soluzione a quella approssimata. Si osservi infatti che i tipi di operazioni di rilassamento possono rigurdare sia il contenuto dei nodi che le relazioni tra loro. Poichè ogni rilassamento dovrebe essere considerato singolarmente e in combinazione con gli altri possibili, il numero delle query rilassato può appesantire il metodo proposto.

Capitolo 2

Tree Matching

2.1 Introduzione

Gli alberi sono una delle più importanti metodologie per la rappresentazione di dati strutturati. Sono in grado di rappresentare qualunque informazione che possieda una struttura gerarchica e possono quindi essere utilizzati indistintamente per rappresentare frasi del linguaggio naturale, formule algebriche o strutture molecolari.

Definizione 2.1 *Un albero radicato è una struttura $T = (V, E, root(T))$, dove V rappesenta un insieme finito di nodi, $root(T)$ appartiene a V ed è il nodo chiamato radice di T, e E è una relazione binaria su V che soddisfa le condizioni sotto riportate. Se $(u, v) \in E$ diremo che (u, v) è un edge e che il nodo u è il padre del nodo v, rappresentando tale relazione con $u = parent(v)$. L'insieme delle edge deve soddisfare le seguenti condizioni:*

1. *La radice non ha padre.*
2. *Ogni nodi ad eccezione della radice ha uno ed un solo padre.*
3. *Tutti i nodi sono raggiungibili seguendo le edge dalla radice*

Gli alberi da noi considerati sono detti **alberi radicati ed etichettati**. Andiamo ora a spiegare il significato di tali aggettivi:

- **radicati:** Significa che nell'albero T esiste un nodo chiamato radice ed indicato con $root(T)$ che è antenato di tutti i nodi di T
- **etichettati:** Significa che ogni nodo possiede ed è caratterizzato da una etichetta che nel nostro caso è una stringa di caratteri

Il confronto fra alberi trova di conseguenza numerose applicazioni in sistemi che gestiscono ed elaborano la conoscenza, dalla biologia molecolare alla visione artificiale, ed altri campi riguardanti la pattern recognition. Lo scopo comune è quello di confrontare un *pattern* (percorso), rappresentato come albero, con modelli, anch'essi rappresentati come alberi (*Tree matching*).

Tree matching significa cercare le istanze o le corrrispondenze di un dato percorso d'albero (*tree pattern*) in un albero obiettivo (*target tree*). Tali corrispondenze sono le soluzioni a problemi chiamati di inclusine d'albero (*Tree Inclusion Problem*). Ciascun tipo di questi problemi si distingue da un altro secondo le regole di similarità che sono richieste tra il tree pattern e il target tree.

Il **Tree Inclusion Problem** più generale prevede, dato un tree pattern P ed un target tree T di localizzare i subtree di T che sono rappresentazioni di P. Questi problemi si dividono in due gruppi: problemi nei quali l'ordine è importante, cioè dove l'ordine da sinistra a destra dei fratelli è fissato, e problemi nei quali non lo è. Molti problemi (come nel campo della genetica per determinare le malattie basate su modelli di alberi genealogici) richiedono di considerare gli alberi non ordinati.

2.2 Unordered tree inclusion problem

La definizione proviene da Kilpeläinen che in [7] descrive tutti i problemi di inclusione d'albero definendoli e proponendo, per alcuni di essi, algoritmi per la loro risoluzione.

La maggioranza dei problemi di inclusine vengono definiti attraverso una funzione di *embedding* che può essere costruita tra i due alberi tree pattern e target tree e mette in relazioni i nodi dell'uno con quelli dell'altro.

Definizione 2.2 ***Unordered tree inclusion problem*** *Siano dati due alberi Q e T. Una funzione iniettiva che mette in relazione i nodi di Q con quelli di T*

è un embedding di Q in T se preserva le label e le ralazioni di parentela. Tale che per ogni nodo $q_i, q_j \in Q$*:*

1. $f(q_i) = f(q_j) \iff q_i = q_j$,
2. $label(q_i) = label(f(q_i))$,
3. q_i *è antenato di* $q_j \iff f(q_i)$ *è antenato di* $f(q_i)$

Un esempio di funzione di embedding per questo problema di inclusione è quella che mette in relazione i nodi evidenziati nella figura qui sotto (2.1).

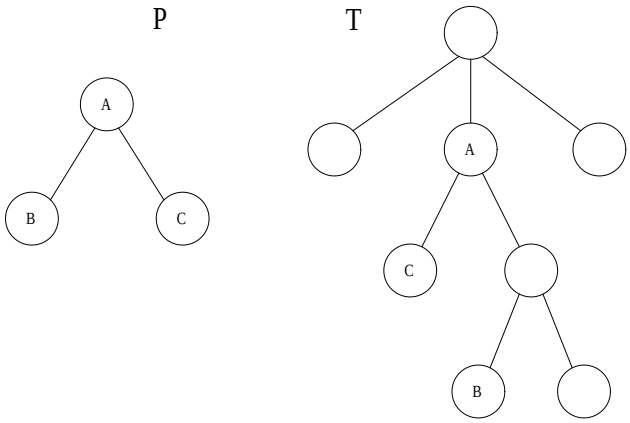


Figura 2.1: Tree inclusion problem

Nell'esempio di fiugura 2.1 l'albero P (tree pattern) risulta incluso in modo unordered nell'albero T (target tree).

2.2.1 Unordered path inclusion problem

Nell'unordered tree inclusion problem la funzione di embedding non vincola i nodi a mantenere tra loro gli stessi legami di parentela. Volendo ricercare nel target tree T un match esatto del tree pattren P occorre definire un altro tipo di problema la cui definizione proviene sempre da Kilpeläinen

Definizione 2.3 ***Unordered path inclusion problem*** *Siano dati due alberi P=(V,E,root(Q)) e T=(W,F,root(Q)). Una funzione iniettiva che mette in relazione i nodi di P con quelli di T è un path embedding di P in T se preserva le label e le relazioni di parentela. Tale che per ogni nodo* $p_i, p_j \in Q$

$(p_i, p_j) \in E$ *se e solo se* $(f(p_i), f(p_j)) \in F$

Prendiamo in considerazione la funzione che mette in relazione i nodi evidenziati nell'esempio della figura seguente (figura 2.2):

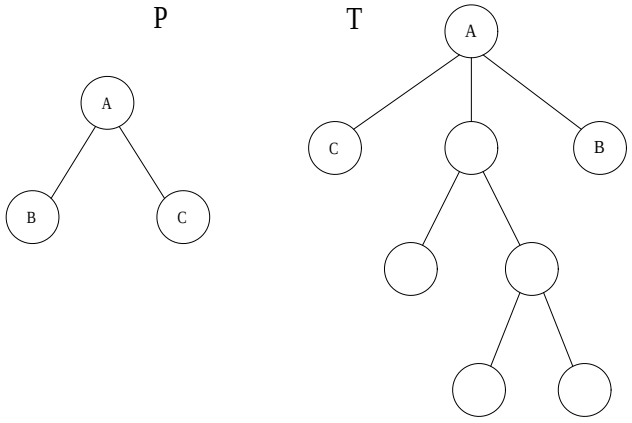


Figura 2.2: Path inclusion problem

In questo caso tutti i legami di parentela fra i nodi sono rimasti invariati nei nodi del target tree.

Possiamo affermare che esiste un unoredered path included tree in T se, operativamente, è possibile ottenere P da T solo attraverso la potatura di nodi di T e la permutazione delle label tra i nodi fratelli rimasti.

2.3 Tree Embedding

Possiamo quindi affermare che, volta deciso come interpretare sia l'albero query che quello dati, il problema di rispondere ad una query può essere ricondotto al problema di ricostruire l'albero query nell'albero dati. Tale problema è conosciuto come *Tree Embedding*.

Il problema dell'*exact ordered tree embedding* è già stato largamente studiato ed è risolvibile in un tempo polinomiale. Noi non siamo però interessati ad alberi ordered siccome l'ordine può essere inconsistente e per questo noi lo ignoreremo. Infatti sarebbe necessario che l'utente nel formulare la query conoscesse l'ordine dei dati nei documenti. Inoltre non siamo interessati al tree embedding esatto quindi per giungere al risultato voluto inizieremo col considerare l'unordered tree inclusion problem.

Come visto precedentemente l'unordered tree inclusion problem definito da Kilpeläinen non garantisce che le relazioni padre figlio tra i nodi del tree pattren

siano preservate dalla funzione f di embedding. Tuttavia, volendo risolvere query approssimate, possiamo decidere di accettare questa funzione f ridefinendola in modo più appropriato come in [12].

Definizione 2.4 ***Tree embedding***

Una funzione f che opera da un insieme Q di nodi appartenenti alla query ad un insieme D di nodi dati è chiamata embedding se per ogni q_i,$q_j \in Q$

1. $f(q_i = f(q_j) \Longleftrightarrow q_i = q_j$,
2. $label(q_i) = label(f(q_i))$,
3. q_i *è padre di* $q_j \Longleftrightarrow f(q_i)$ *è antenato di* $f(q_i)$

Questo rilassamneto del vincolo di parentela padre-figlio in quello più generale di antenato-discendente permette un certo grado di approssimazione nel confronto della struttura ma non produce nessuna metrica in grado di quantificare quale sia il peso dell'approssimazione accettata.

Schieder e Naumann in [12] hanno studiato il problema dell'embedding approssimato per alberi unorderd. Questo tipo di embedding come visto, è riconducibile al tree inclusion problem definito in [7] dove Kilpelainen e Mannila dimostrano che tale problema è NP-completo.

Risolvere una query come proposto in [12] riconducendo il problema al tree inclusion problem è però limitato poichè le soluzioni ritrovate non sono filtrate.

Sarebbe invece opportuno ritrovare solo le risposte che soddisfano a pieno la query proposta dall'utente e tutte quelle che sono simili alla query stessa ma secondo precisi criteri di selezione.

2.3.1 Tree Embedding Approssimato

Il nostro obiettivo è infatti effettuare embed nei quali le label e le relazioni di parentela tra i nodi siano preservate ed un eventuale rilassamento sui vincoli contenuti nel tree pattern P sia debitamente riconosciuto e quindi ben pesato al fine di introdurre una distanza tra P e l'embed trovato che sia una metrica. Solo in questo modo è sucessivamente possibile classificare le soluzioni trovate secondo la distanza con il tree pettern (query).

Per risolvere questo problema è necessario misurare la qualità dell'emmbedding stesso pesando ogni rilassamento effettuato.

Per permettere alla funzione f di embedding di misurare effettivamente la distanza che può esistere fra il tree pattern P ed il target tree T, T.Shilieder in [11] ritorna alla definizione di unordered path inclusion problem (definizione: 2.3)

Propone però una preelaborazione sull'albero query (tree pattern). Il formalismo del tree embedding permette di risolvere solo l'embedding esatto. Per cercare anche tutti i rusultati simili, ricorre all'utilizzo di trasformazioni nella query (**basic query trasformations**). Una basic query trasformation consiste in una modifica alla query che si traduce attraverso l'inserimento, la cancellazione o la modifica della label di un nodo. A differenza dalla definizione di tree edit distance in cui si parla di una qualunque sequeza di edit operation che trasformi un albero nell'altro, in questo caso le query trasformations che possono operare sulla query sono ristrette a solo quelle che generano query che mantengono il significato semantico. Non è, per esempio, permessa la calcellazione di tutte le foglie della query originale poichè sono le foglie che esprimono l'informazione che l'utente sta cercando.

Riportiamo qui sotto le **query trasformation** presentate in [11]

1. **Inserimento:** operazione che trasforma un edge della query in un nodo che possiede edge in ingresso ed in uscita. Osserviamo che questa definizione non permette di aggiungere una nuova radice o nuove foglie.

2. **Cancellazione di un nodo interno:** operazione che rimuove un nodo u interno alla query (non è possibile rimuovere la radice) insieme con l'edge entrante e collega l'edge uscente con il padre di u.

3. **Cancellazione di una foglia:** operazione che rimuove una foglia u insieme con l'edge entrante se e solo se il padre di u ha due o più figli (incluso u) che sono foglie della query.

4. **Modifica di una label:** operazione che cambia la label di un nodo.

Ciascuna basic trasformation ha un **costo** che è assunto essere un numero non negativo. Per assegare tali costi sono possibili diversi criteri:

- Un primo criterio semplicistico protrebbe assegnare ad ogni query trasformations uno stesso costo *costo*

- Un secondo criterio potrebbe distinguere costi diversi per ogni tipo di query trasformations
- Un ulteriore criterio protrebbe richiedere all'utente di assegnare un costo ad ogni tipo di query trasformation da eseguire sulla query

Una volta deciso il criterio di assegnazione dei costi Shlieder passa a definire diversi termini per arrivare alla definizione di *Approximate query matching problem*

Definizione 2.5 ***Transformed query***

Una trasformed query è una query derivata da quella iniziale attraverso una sequenza di basic transformations in cui tutte le cancellazioni precedono tutte le modifiche e tutte le modifiche precedono tutti gli inserimenti.

Definizione 2.6 ***Embedding cost***

L'embedding cost di una transformed query è definito come la somma dei costi di tutte le basic transformations che la hanno prodotta.

Definizione 2.7 ***Query closure***

La chiusura di una query Q è l'insieme di tutte le transformed query che possono essere derivate da Q.

Definizione 2.8 ***Embedding group***

Un embedding group è un insieme di coppie, dove ciascuna coppia è costituita da un embedding ed il suo costo. Tutti gli embedding in un group hanno la stessa radice.

Definizione 2.9 ***Approximate Query matching problem***

Dato un albero T e la chiusura di una query Q, l'approximate query matching problem consiste nell'identificare tutti gli embedding group e rappresentare ogni gruppo attraverso la coppia (u, c) detta ***root-cost-pair*** *, dove u è la radice di tutti gli embedding nel gruppo e c è il costo minore fra tutti gli embedding del gruppo.*

Ciascuna *root-cost-pair* rappresenta un risultato alla query. Un algoritmo, per essere in grado di risolvere l'approximate query matching problem deve cercare

tutti i risultati alla query. Tuttavia, l'utente è generalmente interessato solo ai risultati migliori.

Prosegue quindi con una operazione di ordinamento per ciascuna root-cost-pair tra query ed albero dati, che restituisca gli embedding a partire da quello con costo minore.

L'esempio di interrogazione da loro cosiderato è riportato qui sotto (figura: 2.3).

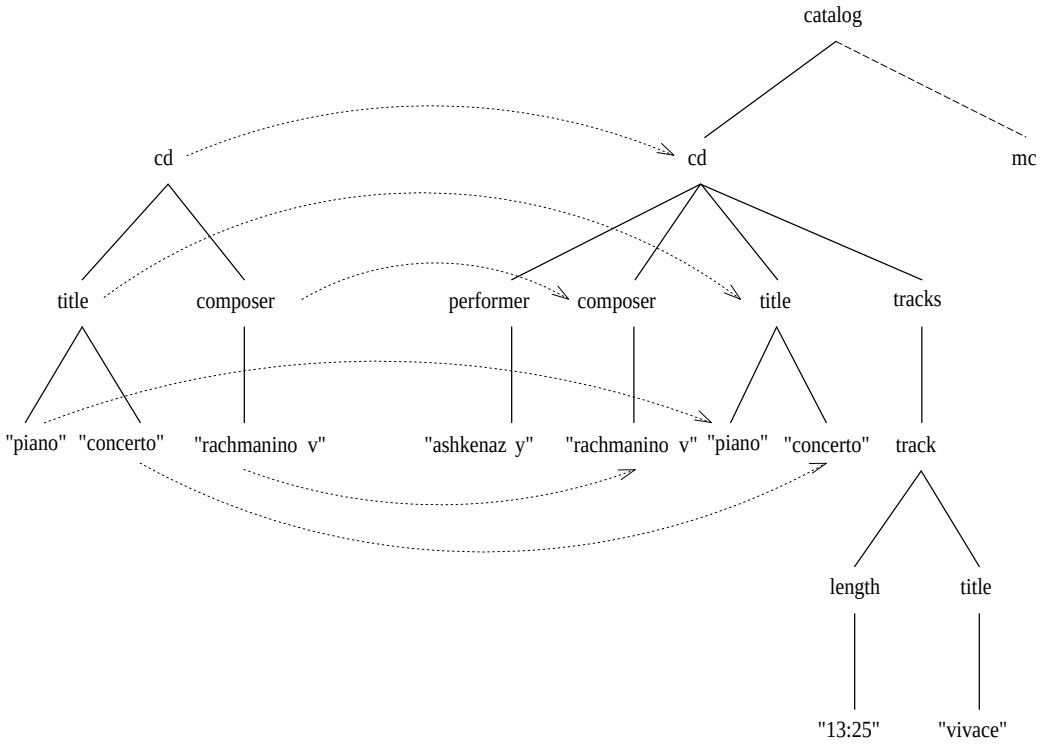


Figura 2.3: Esempio di interrogazione

Noi osserviamo che, sebbene in [11] siano presenti diverse intuizioni come la rappresentazione della struttura degli alberi tramite i suoi nodi dove ognuno è caratterizzato da diversi numeri per mantenere le informazioni sulla struttura stessa dell'albero e i costi relativi ai rilassamenti riguardanti quel nodo, la metodologia proposta per risolvere la query non ci soddisfa. Sia la trasformazione della query qui proposta, sia il rilassamento della query citato al capitolo 1 ([1]) rimangono operazioni indefinite che se affrontate superficialmente non producono alcun risultato e se approfondite possono appesantire gravemente il risolvere la query in modo approssimato.

2.4 Tree edit distance

Per risolvere questi problemi abbiamo bisogno di tecniche efficienti che effettuino match approssimati tra i documenti XML basandosi sulla struttura dell'albero. Ogni volta che chiediamo un match approssimato chiediamo una distanza che sia una *metrica* e che quantifichi efficientemente il match stesso. Tale distanza deve essere in relazione sia ad incosistenze nella struttura che nel contenuto.

Siano S_1 e S_2 due sorgenti dati di documenti XML originate dalla stessa o differenti DTD. Noi stiamo cercando algoritmi in grado di calcolare una operazione di join approssimato tra le sorgenti usando la metrica della tree edit distance. Assumiamo ora che gli interi documenti XML debbano essere confrontati.

Definizione 2.10 ***Approximate TDist Join***

Date due sorgenti di dati XML, S_1, S_2 ed una soglia di distanza τ, sia $TDist(d_1, d_2)$ una funzione che calcola la tree edit distance tra i due documenti $d_1 \in S_1$ e $d_2 \in S_2$. L'operazione di join approssimato tra due sorgenti di dati XML restituisce come risultato tutte le coppie di documenti $(d_1, d_2) \in (S_1 \times S_2)$ tali che $TDist(d_1, d_2) \leq \tau$.

I documenti XML, come già ricordato, sono alberi etichettati e ordinati. Il problema della definizione di una distanza, che sia metrica, tra alberi ordinati ed etichettati è già stato studiato e definito come tree edit distance.

Questa distanza è la naturale generalizzazione della edit distance precedentemente definita per il dominio delle stringhe. Informalmente la tree edit distance tra due alberi è definita come il minor numero di operazioni (inserimento, cancellazione e modifica di un nodo) necessarie per trasformare un albero nell'altro. Sono stati proposti una grande varietà di algoritmi per calcolare la tree edit distance tra alberi lebeled ed ordered [16].

2.5 Preliminari

Sia Σ un alfabeto di dimensione $|\Sigma|$. Sia $\varepsilon \ni \Sigma$ il simbolo nullo. Per una data DTD D, sia Δ l'insieme di tutte le etichette dell'albero ordinate ben formate sotto D. Senza perdere in generalità noi possiamo assumere che tutti i nodi di D tanto quanto gli elementi siano definiti su Σ.

Nel dominio delle stringhe, la nozione di edit distance è già stata largamente utilizzata per quantificare le differenze tra stringhe.

Definizione 2.11 ***Edit distance***

L'edit distance tra due stringhe σ_1, σ_2 $ed(\sigma_1, \sigma_2)$ è definita come il minor numero di operazioni (inserimento, cancellazione e sostituzione) su un singolo carattere richieste per trasformare una stringa in un altra.

Date due stringhe σ_1, σ_2 esiste un algoritmo molto conosciuto [10] basato sulla programmazione dinamica in grado di calcolare la edit distance tra loro in $O(|\sigma_1||\sigma_2|)$ nel tempo e nello spazio. Le edit operations sono assunte con costo unitario, ma sono state proposte alternative in cui il costo ha valori diversi o in cui esistono diverse operazioni di edit. Comunque la funzione ed() rimane una metrica.

Una generalizzazione della edit distance fra stringhe è la nozione di *tree edit distance* definita allo scopo di quantificare le differenze tra una coppia di alberi ordinati.

Siano T_1, T_2 due documenti XML ben formati sotto la stessa o differenti DTD. Possono essere concettualmente rappresentati (dopo una operazione di parsing) come alberi ordinati ed etichettati dove le etichette dei nodi contengono gli elementi dei tag, valori PCDATA, nome degli attributi o valore degli attributi e l'annidamento degli elementi è espresso dalla struttura dell'albero.

Definizione 2.12 ***Tree edit distance***

Dati due alberi T_1, T_2 la tree edit distance tra loro, indicata con TDist(T_1,T_2), è definita la sequenza a costo minore di edit operations (inserimneto, cancellazione, sostituzione etichetta) su un singolo nodo dell'albero, necessarie per trasforamre un albero nell'altro.

Come sopra premesso ci sono tre tipi di edit operations e sono inserimento cancellazione e sostituzione e di un nodo. Sostituire un nodo n significa cambiare l'etichetta al nodo n stesso. Cancellare un nodo n significa che tutti i figli di n diventano figli del padre di n e in seguito rimuovere il nodo n stesso. Inserire un nodo n è l'operazione complementare alla cancellazione. Significa che inserendo n come figlio del nodo m, un sottoinsieme dei figli di m possono diventare figli del nodo n.

Rappresentando le edit-operations come $a \longrightarrow b$, dove a rappresenta Λ (nodo nullo) o un nodo dell'albero T_1 e b rappresenta Λ o un nodo dell'albero T_2,

indicheremo l'operazione di sostituzione della label se $a \neq \Lambda$ e $b \neq \Lambda$, l'operazione di inserimento se $a = \Lambda$ e $b \neq \Lambda$ e l'operazione di cancellazione se $a \neq \Lambda$ e $b = \Lambda$. Sia T_2 l'albero che risulta dopo aver applicato a T_1 la edit operation $a \longrightarrow b$ all'albero T_1 allora sciveremo $T_1 \Longrightarrow T_2$ attraverso $a \longrightarrow b$.

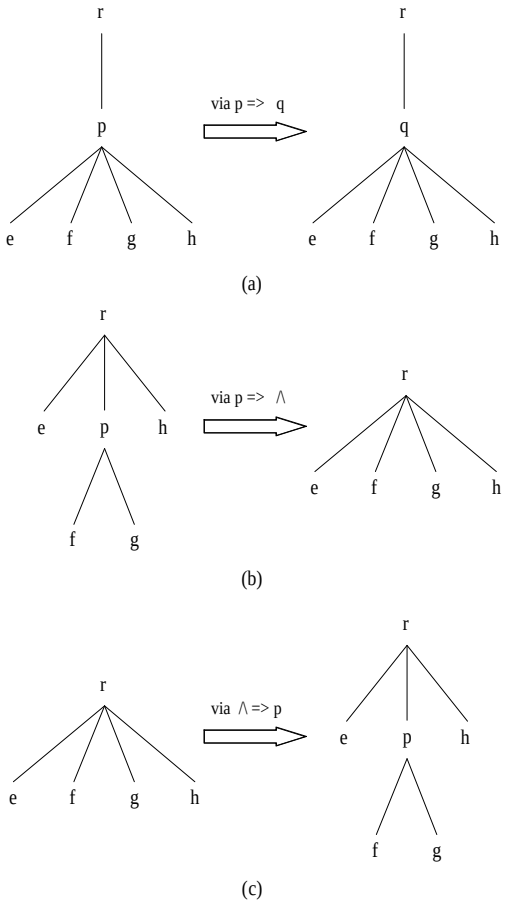


Figura 2.4: Esempi di edit operations

Nella figura 2.4 è riportato un esempio per ognuna delle edit operations. In `(a)` si rinomina la label `p` e viene cambiata in `q`. In `(b)` attraverso la cancellazione del nodo `p` tutti i suoi nodi figli diventano figli del padre di `p`. Infine in `(c)` viene inserito un nodo `p` e un sottoinsieme dei nodi figli di `r` diventano figli del nodo appena inserito `p` (in questo caso `e` e `h`).

Sia S una sequenza $e_1, e_2..., e_k$ di edit operations. Una **S-derivation** dall'albero T all'albero T' è una sequenza di alberi $T_0, ..., T_k$ dove $T_0 = T$ e $T_k = T'$ e $T_{i-1} \Longrightarrow T_i$ attraverso e_i per ogni $1 \leq i \leq k$.

Sia γ una funzione di costo che assegna a ciascuna edit operations $a \longrightarrow a$ un numero reale non negativo $\gamma(a \longrightarrow b)$. Imponiamo che γ sia una distanza metrica tale per cui valgano quindi le seguenti proprietà:

1. $\gamma(a \longrightarrow a) \geq 0, \gamma(a \longrightarrow a) = 0$
2. $\gamma(a \longrightarrow b) = \gamma(b \longrightarrow a)$
3. $\gamma(a \longrightarrow c) \leq \gamma(a \longrightarrow b) + \gamma(b \longrightarrow c)$

Si può estendere la funzione di costo γ alla sequenza S di edit operations ponendo:

$$\gamma(S) = \sum_{i=1}^{|S|} \gamma(e_i)$$

Possiamo ora definire formalmente la distanzza (Tree Edit Distance) tra due alberi T e T'.

Definizione 2.13 ***Tree edit distance***

$\delta(T, T') = \min_S$ *($\gamma(S)$ | S è una sequenza di edit operations che trasforma T in T')*

Quindi dando un costo a queste edit operation (dove tale costo potrebbe dipendere dai nodi in questione), la distanza tra due alberi è il minore tra i costi delle sequenze di operazioni che trasformano un albero nell'altro.

Osserviamo inoltre che δ è una distanza metrica in accordo con la definizione di γ.

2.6 Tree edit distance fra alberi unordered

L'algoritmo citato nel paragrafo precedente proposto da Zhang e Shasha in [16], calcola la distanza fra alberi ordered.

Dato un albero T sia $h(T)$ la sua altezza. Dati due alberi T_1, T_2, tale algoritmo è in grado di calcolare la tree edit distance tra loro con una complessità pari a $O(|T_1||T_2||h(T_1)||h(T_2)|$ nel tempo e in $O(|T_1||T_2|)$ nello spazio.

Tuttavia, come detto noi siamo interessati a determinare la distanza fra alberi unordered. Vogliamo cioè calcolare la distanza fra alberi in cui l'ordine tra fratelli non è rilevante.

Nessuno degli algoritmi sopra citati può essere utilizzato. Inoltre, come riportato in [16], [7] e [9] calcolare l'edit distance tra alberi unordered risulta np-completo.

Esiste, tuttavia, la proposta di un algoritmo per il calcolo della tree edit distance tra alberi unordered. Questa proposta è contenuta in [13] dove gli autori forniscono sia algoritmi esaustivi che evidenziano la complessità del problema sia algoritmi approssimati per limitare tale complessità.

Ritenendo significativa la proposta la riportiamo di seguito per intero premettendo che ad essa faremo riferimento nel progettare l'algoritmo per il calcolo della tree edit distance presentato nel capitolo 3.

2.6.1 Numerazione dei nodi di un albero

Dato un albero è spesso conveniente utilizzare un numero per riferirsi ai suoi nodi. Per un albero ordinato è possibile utilizzare indistintamente la numerazione *postorder* o *preorder*, mentre per alberi disordinati può andar bene anche la numerazione *unorder*.

Se vogliamo produrre un elenco degli elementi inseriti in un albero dobbiamo compiere un attraversamento dello stesso albero (tree taversal o tree walk) che ci permetta di visitare tutti i suoi nodi. I più importanti tree traversal sono denominati inorder, preorder e postorder e ad ognuno di essi è associato l'omonimo elenco ordinato dei suoi nodi. Per questi tre tipi di tree traversal otteniamo lo stesso elenco di nodi nei casi di albero vuoto ed albero costituito da un singolo nodo.

In questa tesi, d'ora in avanti, faremo riferimento solo ad alberi con radice, quindi sottointederemo il riferimento alla radice r rappresentando il simbolo dell'albero (T, r) con il solo simbolo T.

Spieghiamo in termini ricorsivi come viene ricavato l'elenco ordinato dei nodi. Consideriamo l'albero (T, n) in figura 2.5 dove $T_1, T_2 ... T_k$ rappresentano i sottoalberi.

Definizione 2.14 ***Preorder ordering***

L'elenco secondo l'ordinamento preorder dei nodi di T è dato dalla radice n, seguita dai nodi di T_1 in preorder ordering, seguito dai nodi di T_2 in preorder ordering e così via...

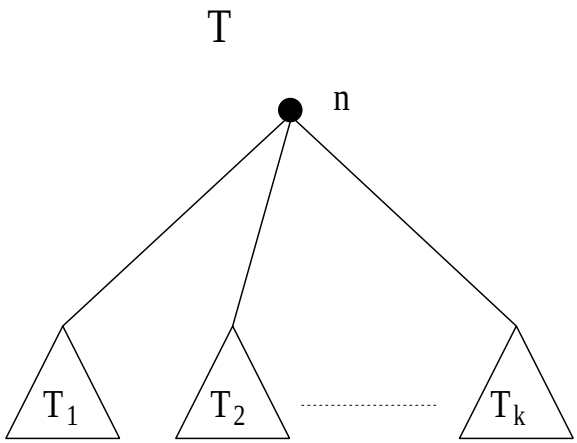


Figura 2.5: Rapprsentazione di un albero mediante i suoi sottoalberi

Definizione 2.15 ***Postorder ordering***

L'elenco secondo l'ordinamento postorder dei nodi di T è dato dai nodi di T_1 *in postorder ordering, seguito dai nodi di* T_2 *in postorder ordering e così via ..., seguito da infine dal nodo n.*

Definizione 2.16 ***Inorder ordering***

L'elenco secondo l'ordianmento inorder dei nodi di T è dato dai nodi di T_1 *in inorder ordering, seguito da dal nodo n , seguito dai nodi di* T_2 *in inorder ordering e così via... In genere viene utilizzato negli alberi binari.*

2.7 Mapping

Gli autori iniziano con l'introdurre l'operazione di *mapping*. Supponiamo di avere assegnato ai nodi dell'albero una numerazione in postorder ordering. Sia $T[i]$ l'i-esimo nodo dell'albero T nel dato ordine.

Sia TreeDist(T_1,T_2) la funzione che esegue l'algoritmo per il calcolo della Tree Edit Distance (TDist) tra glia alberi T_1 e T_2. Tale algoritmo deve trovare un *mapping* M tra i nodi dei due alberi; il mapping detterà tutte le relazioni tra i nodi dei due alberi. L'obiettivo dell'algoritmo sarà calcolare il mapping con il minimo costo secondo uno stesso modello di costo scelto.

L'algoritmo costruisce un mapping tra i nodi dei due alberi nel quale sono formulate tutte le corrispondenze fra di loro.

Tramite l'operazione di mapping si può rappresentare graficamente l'applicazione delle edit operations sui nodi dei due alberi. Il mapping in figura 2.6 mostra la trasformazione dell'albero T nell'albero T' corrispondente alla sequenza di operazioni:

- Cancellazione del nodo d
- Inserimento del nodo f

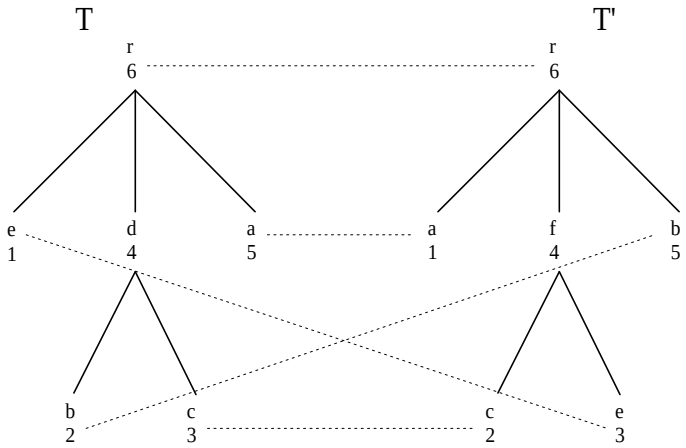


Figura 2.6: Rappresentazione di un mapping

Le lettere relative a ciascun nodo indicano la label del nodo mentre i numeri indicano la posizione dei nodi nell'albero secondo l'ordinamento postorder. Le linee tratteggiate da un nodo u di T ad uno v di T' indicano che u sarà trasformato in v se $u \neq v$, oppure che u e v non cambiano se $u = v$.

Un mapping M da un albero T ad uno T' è una terna (M, T, T') o più semplicemente M se non genera confusione, dove M è un insieme di coppie di interi (i,j) che soddisfano i seguenti vincoli:

1. $1 \leq i \leq |T|, 1 \leq j \leq |T'|$, dove $|.|$ rappresenta il numero di nodi dell'albero indicato
2. Per ogni coppia (i_1, j_1) e (i_2, j_2) in M
   (a) $i_1 = i_2$ se e solo se $j_1 = j_2$ (uno ad uno)
   (b) T_{i_1} è un antenato di T_{i_2} se e solo se T'_{j_1} è un antenato di T'_{j_2} (**Ancestor Relatioship Preserved**)

Sia M il mapping da T a T'. Siano $I \subset V(T)$ e $J \subset V(T')$ gli insiemi dei nodi che non comapaiono, rispettivamente come primo e secondo termine, nelle coppie dell'insieme M. Allora possiamo definire il costo ci un mapping M, usando le edit operations, come:

$$\gamma(M) = \sum_{(i,j)\in M} \gamma(T[i] \longrightarrow T'[j]) + \sum_{i\in I} \gamma(T[i] \longrightarrow \Lambda) + \sum_{j\in J} \gamma(\Lambda \longrightarrow T'[j])$$

Ossia il costo del mapping è dato dal costo:

- Dell'operazione di cambiamento del nodo $T[i]$ in $T'[j]$ per tutte le coppie $(i, j) \in M$, cioè per tutti i nodi che nella rappresentazione grafica sono toccati dalla linea.
- Dell'operazione di cancellazione dei nodi di T non toccati da linee tratteggiate.
- Dell'operazione di inserimento dei nodi di T' non toccati di linee tratteggiate.

Il mapping gode di diverse proprietà fra cui la composizione.

Enunciato 2.1 *Sia M un mapping da T a T' e sia M' un mapping da T' a T". Definiamo M composto M', indicato col simbolo* $M \circ M'$*, l'insieme delle coppie (i,j), dove* $i \in V(T) e j \in V(T'')$*, per cui esiste un nodo* $k \in V(T')$ *che è collegato a i e a j con delle linee tratteggiate rappresentative rispettivamente dei mappin M e M'. Formalmente:*

$M \circ M' = \{(i, j) | \exists k$ *tale per cui* $(i, k) \in M e (k, j) \in M'\}$

Il seguente enunciato è dimostrato in [13]

Enunciato 2.2 *Dato il mapping* $M \circ M'$*, allora* $\gamma(M \circ M') \leq \gamma(M) + \gamma(M')$

Diamo un uleriore lemma che metta in relazione le sequenze di edit operations e i mapping:

Enunciato 2.3 *Sia S una sequenza* $e_1, e_2, ..., e_k$ *di edit operations da T a T', allora esiste un mapping M da T a T' tale che* $\gamma(M) \leq \gamma(S)$*. D'altro canto, per ogni mapping M esiste una sequenza di edit operations S tale che* $\gamma(S) = \gamma(M)$*.*

Quindi otteniamo che:
$\delta(T, T') = \min_M \{\gamma(M) | M$ è un mapping che trasforma T in $T'\}$

2.8 Marking

Il primo problema da affrontare per calcolare la tree edit distance è quello di trovare due insiemi di nodi, appartenenti ai due alberi da confrontare, tali che una volta eliminati producano due alberi isomorfi.

Definizione 2.17 ***Isomorfismo fra due alberi***

Due alberi A e B sono isomorfi se esiste un mapping (M,A,B) uno a uno tale che ogni relazione padre figlio in A è mantenuta in B.

Definizione 2.18 ***Insieme Head(A)***

Sia A un albero unordered, possiamo cosiderare che i suoi nodi siano numerati da sinistra a destra secondo il postorder ordering. Sia quindi A[i] l'i-esimo nodo in tale numerazione, leaves(A) il numero di foglie presenti nell'albero A, par(n) il padre del nodo n e deg(n) il numero di figli di n. Allora possiamo definire:

Head(A) = { n | (n è la radice di A o (deg(par(n)) > 1 }

Per ciascun $n \in Head(A)$, sia s(n) la stringa contenente le label dei nodi che partono da n sino al primo nodo che ha più di un figlio o è una foglia. Tale stringa potrebbe contenere anche solo un nodo.

Se si sostituiscono tutti i nodi in A che costituiscono la stringa $s(n)$ col nodo n in cui la label sostituita dalla stringa $s(n)$ si ottiene un albero in cui ciascun nodo ha più di un figlio.

Definizione 2.19 ***Marking***

Dati due alberi T e T', un marking K è rappresentato da una coppia $(S_T, S_{T'})$ dove S_T è un insieme di nodi in T e $S_{T'}$ è un insieme di nodi in T'.

Definizione 2.20 ***Marked tree***

Dato un albero T e un marking K, si definisce Marked Tree di T, indicandolo con K(T) l'albero ottenuto cancellando i nodi di T corrispondenti all'insieme S_T del marking K.

Definizione 2.21 ***Reduced marked tree***

Dato un albero T ed un marking K, si definisce Reduced Marked Tree di T, indicandolo con RK(T), l'albero ottenuto sostituendo a ciascuna stringa s(n) in K(T) dove $n \in Head(K(T))$ un nodo n

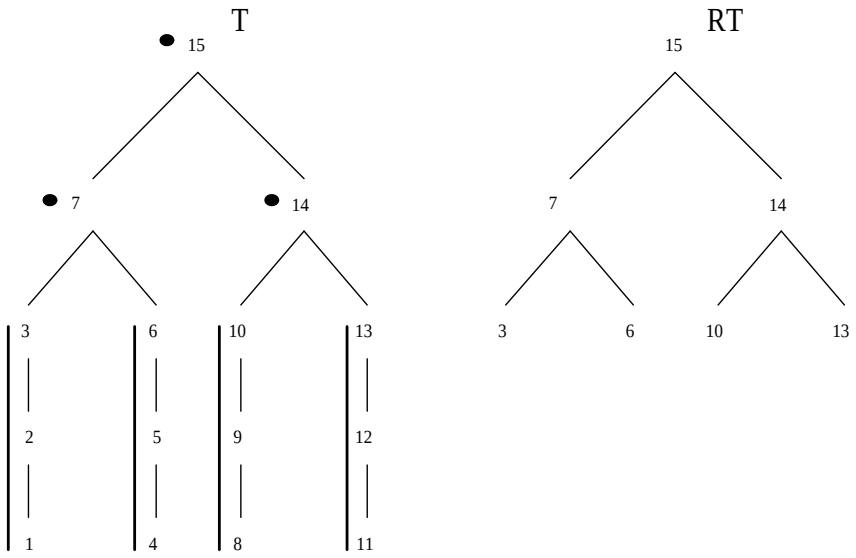


Figura 2.7: Riduzione di un albero

Definizione 2.22 ***Marking legale***

Siano dati due alberi T e T' e K un marking K, diremo che K è un marking legale se RK(T) è isomorfo a RK(T') cioè se i nodi cancellati dai due alberi producono alberi con la stessa struttura.

Possono esistere diversi isomorfismi tra RK(T) e RK(T') e ciascuno di questi isomorfismi è chiamato complete mapping.

Definizione 2.23 ***Complete mapping***

Dati due alberi isomorfi T e T', definiamo come complete mapping CM i mapping (M,T,T') tali che per ogni $i \in T$ e $j \in T'$ esiste una coppia $(i,j) \in M$. Questo significa che nella rappresentazione grafica le linee del complete mapping devono toccare tutti i nodi di T e T'.

Il costo di un complete mapping (CM) denotato con $cost(CM)$ è definito come:

$$cost(CM) = \sum_{(i,j) \in CM} ndist(RK(T)[i], RK(T')[j])$$

dove $ndist()$ è la funzione per determinare la distanza tra due nodi appartenenti agli alberi ridotti $RK(T)[i]$ e $RK(T')[j]$.

Possiamo ora definire il costo di un marking legale

Definizione 2.24 ***Costo di un markig legale***

Dati due alberi T e T' e un marking K legale si definisce costo del marking la somma del costo del miglior complete mapping più il costo di cancellazione dei nodi in S_T più quello per i nodi in $S_{T'}$.

Formalmente:

$$cost(K) = \sum_{s \in S_T} sdist(s, 0) + \sum_{s' \in S_{T'}} sdist(0, s') + RK\_tdist(|RK(T)|, |RK(T')|)$$

La distanza fra due alberi T e T' sarà data dal minore tra i costi di tutti i possibili marking legali esistenti tra i due alberi. Formalmente possiamo ridefinire la tree edit distance come segue.

Definizione 2.25 ***Tree edit distance***

$\delta(T, T') = \min_K\{$ *costo del marking legale* K *tra* T *e* $T'\}$

L'algoritmo calcola quindi il minor costo tra quelli dei complete mapping ottenibili dai possibili marking legali.

2.9 Costo di un marking legale

Per trovare il miglior complete mapping fra K(T) e K(T') dato il markig K, si procede in modo ricorsivo partendo dal confronto delle radici e scandendo i marking di livello in livello sino ad arrivare alle foglie.

Sia $RK_tdist(i, j)$ la distanza tra il sottoalbero radicato in $RK(T)[i]$ e il sottoalbero radicato in $RK(T')[j]$. Nel calcolo i due nodi $RK(T)[i]$ e $RK(T')[j]$ sono allo stesso livello e si possono presentare i due casi seguenti:

- $deg(RK(T)[i]) \neq deg(RK(T')[j])$ In questo caso $RK_tdist(i, j) = \infty$ poichè la coppia (i,j) non appartiene a nessum complete mapping da $RK(T)$ a $RK(T')$.
- $deg(RK(T)[i]) = deg(RK(T')[j])$ La coppia (i, j) appartiene a tutti i complete mapping da $K(T)$ a $K(T')$ come conseguenza alla condizione di preservazione della relazione di parentela (*ancestor relationship preserved*). Per cercare il migliore mapping tra i figli di $RK(T)[i]$ e quelli di $RK(T')[j]$ si ricorre alla costruzione di un *grafo bipartito.* pesato.

2.10 Grafo Bipartito

Siano $U = i_1, ..., i_n$ e $V = j_1, ..., j_n$ due insiemi disgiunti di nodi appartenenti ai due alberi. In particolare siano i_k e j_k $1 \leq k \leq n$ rispettivamente i figli di $RK(T)[i]$ e di $RK(T')[j]$.

Siano $u \in U$ e $v \in V$. Si assegna a ciascun linea (u, v) del grafo un costo (peso) dove $cost(u, v) =$

- $RK_tdist(u, v)$ se $deg(u) = deg(v)$
- ∞ *altrimenti*

In questo modo il problema di determinare la distanza $RK_tdist(i, j)$ si riconduce alla ricerca della miglior rappresentazione completa nel grafo bipartito (BG)

Come conseguenza si ha che

1. se $deg(RK(T)[i]) \neq deg(RK(T')[j])$ allora $RK_tdist(i, j) = \infty$
2. se $deg(RK(T)[i]) = deg(RK(T')[j]) = 0$ allora
   $RK_tdist(i, j) = ndist(RK(T)[i], RK(T')[j])$
3. se $deg(RK(T)[i]) = deg(RK(T')[j]) \neq 0$ allora
   $RK_tdist(i, j) = ndist(RK(T)[i], RK(T')[j]) + \sum_{(u,v)\in Ma} cost((u, v))$

dove Ma è la rappresentazione completa ottimale nel grafo bipartito costruito.

Supponiamo che nell'esempio (figuara 2.8) riportato le distanze e quindi i costi tra i nodi $RK(T)[2]$, $RK(T)[3]$, $RK(T')[2]$, $RK(T')[3]$ valgano:

- $ndist(RK(T)[2], RK(T')[2]) = 3$
- $ndist(RK(T)[2], RK(T')[3]) = 3$
- $ndist(RK(T)[3], RK(T')[2]) = 3$
- $ndist(RK(T)[3], RK(T')[3]) = 0$

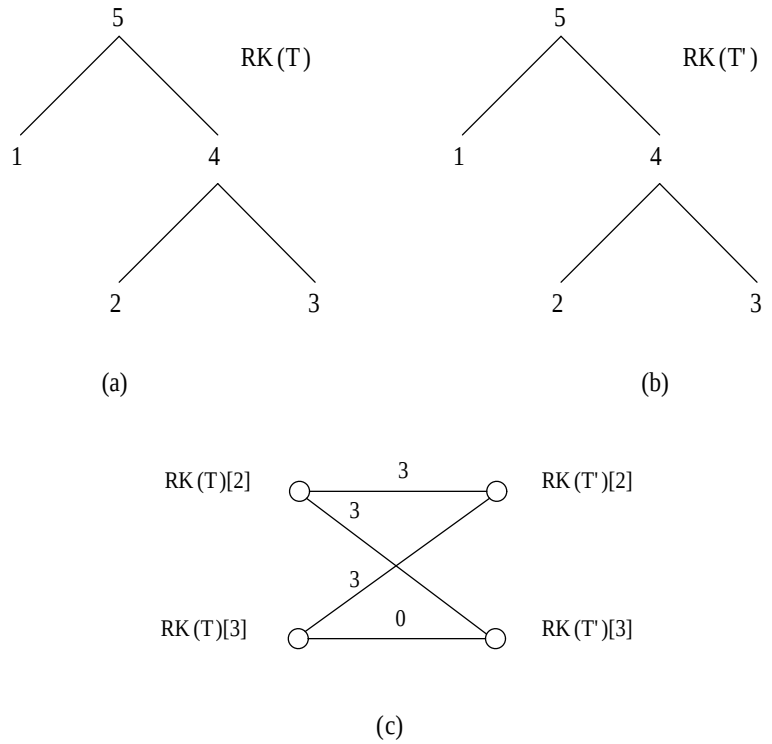


Figura 2.8: Grafo bipartito

Nell'esempio della figura 2.8 si osserva allora, che la rappresentazione completa ottimale del grafo bipartito (c) è quella che associa al nodo 2 di T il nodo 2 di T' e quindi il nodo 3 di T al nodo 3 di T'.

In questo caso infatti, associando al nodo 2 di T il nodo 2 di T' con costo pari a 3 e quindi il nodo 3 di T al nodo 3 di T' con costo pari a 0, si ottiene il costo minore. Nell'altra combinazione possibile (rispettivamente 2 con 3 e 3 con 2) il costo complessivo sarebbe stato di 6 e quindi maggiore.

2.11 Ricerca esaustiva del miglior marking

Per essere certi di trovare il markig che minimizza il costo occorre costruire tutte le combinazioni di marking possibili. Questa operazione, comporta una elevata complessità computazionale in quanto, il numero di reduced tree per un albero di n nodi è dato da $2^n - 1$.

D. Shasha e gli altri autori, in [13] propongono diversi algoritmi euristici per ridurre tale complessità, tutti basati sulla definizione di *random walks*.

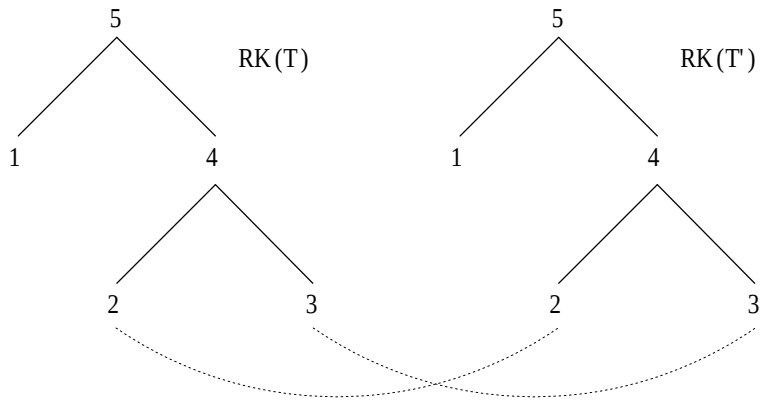


Figura 2.9: Mapping tra alberi ridotti dopo BG

Considerano ciascun marking legale come uno stato e trasformano il problema di calcolare $\delta(T, T')$ in una ricerca nello spazio degli stati. Sia $K = (S_T, S_{T'})$ uno stato. Uno stato $K' = (S'_T, S'_{T'})$ è raggiungibile dallo stato K, indicato con $K \Rightarrow K'$ se una delle seguenti proprietà è verificata:

1. S'_T è ottenuto rimuovendo (o aggiungendo) una stringa da S_T
2. $S'_{T'}$ è ottenuto rimuovendo (o aggiungendo) una stringa da $S_{T'}$

Sotto tale proprietà affermano che K' e un vicino (neighbor) di K. Riportano poi due enunciati di cui noi non riportiamo la dimostrazione.

Enunciato 2.4 *Dato uno stato $K = (S_T, S_{T'})$ il numero dei suoi vicini è limitato da $O(leaves(T) \times leaves(T'))$*

Enunciato 2.5 *Dati due stati $K = (S_T, S_{T'})$ e $K' = (S'_T, S'_{T'})$ esiste una sequenza di stati $K_0, K_1, ..., K_k$ tali che $K_0 = K$, $K_k = K'$ e $K_{i-1} \Rightarrow K_i$ per $1 \leq i \leq k$.*

Uno spostamento da un stato all'altro è chiamato *uphill* (rispettivamente *downhill*) se il costo dello stato di partenza è piu basso (rispettivamente più alto) del costo dello stato destinazione.

Uno stato K è definito come *minimo locale* se in tutti i percorsi che partono dallo stato K, qualunque spostamento downhill avviene sucessivamente ad uno spostamento uphill. Uno stato K è definito come *minimo globale* se possiede il costo più basso fra tutti gli stati.

Sotto queste condizioni Shasha può quindi affermare che:

$\delta(T, T') =$ Costo del minimo globale nello spazio degli stati.

Per la ricerca di tale minimo globale sono poi presentati tre algoritmi approssimati dei quali noi riporteremo solo il nome ed i riferimenti.

1. Iterative improvement
2. Simulated annealing
3. Two phase heuristic

Tuttavia nessuna di queste proposte garantisce di ritrovare il miglior marking legale, cioè quello con costo minore che equivale alla effettiva tree edit distance tra i due alberi.

Parte II

Approccio alla risoluzione di query XML

Capitolo 3

Tree edit distance

In questo capitolo ci si concentrerà sull'approccio seguito per realizzare quella che è la funzione alla base di tutto il progetto: il calcolo della distanza tra alberi. Essa merita una descrizione dettagliata e verrà analizzata preliminarmente anche perchè, come si è visto nella prima parte, riguarda un argomento di ricerca di per sè di notevole interesse per svariati ambiti.

Nel capitolo seguente, si descriverà invece come, prendendo spunto dagli altri tipi di approcci al problema del Tree Matching, si sia realizzazto un filtro per limitare la complessità dell' algoritmo che presenteremo in questo capitolo, sia nel tempo che nello spazio.

Si considererà poi l'algoritmo nel suo insieme e si analizzeranno le ulteriori funzionalità dell'intero lavoro, collocando quanto precedentemente descritto nel contesto globale e mostrando quelle caratteristiche aggiuntive che lo rendono un sistema completo.

3.1 Premesse

3.1.1 Le finalità

Il primo obiettivo da noi perseguito è quello di realizzare un algoritmo che sia in grado di misurare la distanza tra alberi. Come affermato nel capitolo 2 diverese metodologie sono state proposte a tale scopo, tuttavia noi riteniamo che la sola in grado di calcolare una distanza che sia una metrica sia quella che ricorre ad un algoritmo per calcolare la tree edit distance tra alberi etichettati ed unordered.

Progettando tali algoritmi si è cercato di creare dei procedimenti in grado di soddisfare i seguenti requisiti:

- *Rigorosità*: basarsi su una solida struttura teorica, che fornisca un metodo chiaro ed univoco per quantificare la *somiglianza* o *dissimilianza* tra alberi.
- *Praticità*: non utilizzare informazioni che vanno oltre il contemuto dei nodi degli alberi in esame, evitando di introdurne di esterne, non pratiche da ottenere.
- *Flessibilità*: permettere all'utente di esprimere le proprie preferenze fornendogli la possibilità di agire sui parametri che influenzano il calcolo della distanza tra gli alberi.
- *Efficacia*: produrre dei risultati effettivamente utili e di qualità.
- *Efficienza*: essere ottimizzati anche nei riguardi dei tempi di esecuzione, indispensabile per poter gestire notevoli volumi di dati.

Per soddisfare ognuno di questi punti, si è studiato e implementato un procedimento basato, nelle sue linee di fondo, sul concetto di *Tree Edit Distance*. Il procedimento di risoluzione della query vero e proprio è stato suddiviso in più fasi, ognuna implementata singolormente e quindi indipendente dall'altra. Inoltre, per migliorare l'efficienza, si utilizzano una particolare struttura dati (dizionario) ed un *filtro*.

A seconda delle esigenze dell'utente, è possibile ricercare le similarità tra alberi variando l'importanza dei rilassamenti possibili sulla struttura logica del documento. Come si vedrà, è pertanto possibile agire sui principali parametri in modo, ad esempio, da escludere risposte alla query che non presentino la stessa struttura logica della query o che siano *diverse* dalla query stessa oltre una certa soglia.

3.1.2 Rappresentazione degli alberi

Come detto, gli alberi sono la metodologia di rappresentazione scelta sia per il documento contenente i dati che per quello contenente la query. Ciascun elemento di questi documenti XML, che sia valore o attributo è, in tale rappresentazione, espresso nell'albero da un nodo. Il nome del documento sarà la radice

dell'albero. Le relazioni tra i vari elementi nel documento saranno mantenute nell'albero sotto forma di relazioni fra i nodi. Ricordando la definizione 2.1 tali relazioni saranno tutte contenute in E.

Tuttavia la rappresentazione dell'albero nella forma citata poco si presta ad essere interpretata sia da parte di un utente che da parte di un elaboratore.

La forma grafica di un albero esprime molto meglio le relazioni che esistono fra i vari nodi e risulta per un utente di più immediata interpretazione. È subito possibile riconoscere le relazioni di parentela padre-figlio senza dover scorrere la lista delle relazione contenute in E. Per ogni nodo si identificano subito i nodi che sono suoi figli, suoi nipoti e così via. Osserviamo quindi che la grande capacità rapprensentativa della forma grafica è quella di esprimere le parentele in forma di inclusione.

Possiamo infatti affermare che un albero possiede una struttura con annidamenti, cioè ciascun nodo che non sia una foglia, contiene al suo interno i nodi discendenti.

Come suggerito da Philip N. Klein in [8] abbiamo quindi scelto di rappresentare la truttura di un albero non tramite un insieme di relazioni tra i nodi ma concentrando tutte le informazioni in una forma che chiameremo *forma a stringa* rappresenatativa di un albero, dove l'albero viene rappresentato appunto, senza perdita di informazioni, tramite una stringa.

Tuttavia anche tale struttura, pur essendo coincisa e di più immediata interpretazione si presta poco ad analisi e ad elaborazioni da parte di un calcolatore. È necessario scomporre la struttura dell'albero preservando le relazioni tra i nodi. Poichè l'obiettivo è dare la struttura dell'albero in pasto ad un algoritmo che la elabori, la struttura più semplice è quella di un Array.

Abbiamo quindi scelto di rappresentare un albero tramite una lista di nodi, in cui ciascun elemento nodo della lista viene arricchito da informazioni per mantenere traccia della struttura dell'albero stesso. Assegneremo a ciacun nodo u cinque valori:

I primi tre sono necessari a mantenere le informazioni espresse dall'albero

1. **Label(u):** label del nodo u,

2. **Postorder(u)** che rappresenta il numero di postorder assegnato al nodo dell'albero in postorder ordering,

3. **Leaf(u):** che rappresenta il numero di postorder assegnato alla foglia più a sinistra del sottoalbero che ha come radice u.

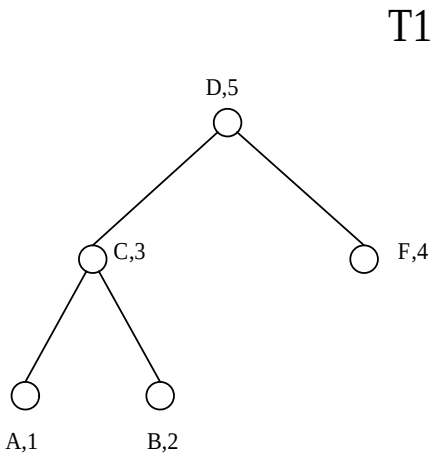
Gli ultimi due numeri hanno un ruolo secondario poichè non necessari ma utili al nostro problema del calcolo della tree edit distance .

1. **Deep(u):** che rappresenta la profondità del nodo u nell'albero assumendo che la radice abbia profondità 0 che sia crescente seguendo la discendenza dalla radice stessa,

2. **Del_cost(u):** costo di cancellazione del nodo.

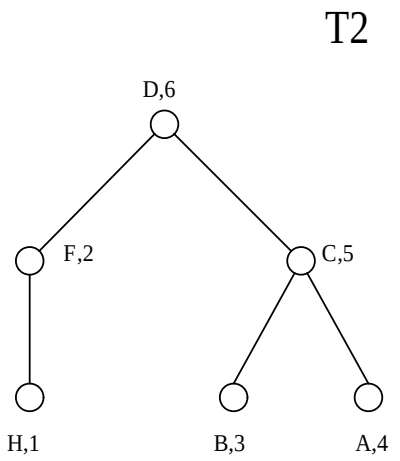
È facile verificare che le informazioni contenute in E sono tutte mantenute nei primi tre numeri assegnati a ciascun nodo. Il numero deep facilita il riconoscimento delle relazioni di parentela padre-figlio e sarà di grande utilità quando andremo a selezionare nell'albero dati i gruppi di nodi da confrontare con l'albero query. Il numero del_cost è invece necessario per aggiungere significato all'operazione di cancellazione di un nodo appartenente all'albero query nel quale non ha senso considerare tutti i nodi con il medesimo costo.

Di seguito sono riportati due esmpi che mostrano come è possibile esprimere tutte le informazioni contenute in un albero attraverso la struttura prima descritta.



Postorder	Label	Leaf	Deep	Del_cost
1	A	1	2	1
2	B	2	2	1
3	C	1	1	1
4	F	4	1	1
5	D	1	0	1

Figura 3.1: Esempio di albero: T1



Postorder	Label	Leaf	Deep	Del_cost
1	H	1	2	1
2	F	1	1	1
3	B	3	2	1
4	A	4	2	1
5	C	3	1	1
6	D	1	0	1

Figura 3.2: Esempio di albero: T2

La prima relazione per la quale è necessario verificare che sia mantenuta è quella di parentela tra nodi.

Dati due nodi u e v, possiamo facilmente verificare se un u è un antenato di v attraverso le condizioni:

Per ogni $u, v \in U$, u è un antenato di v se:

1. $post(u) > post(v)$
2. $leaf(u) \leq post(v)$

Verificando tutte le relazioni di parentela o sfruttando l'ordinamento imposto per la lista dei nodi è possibile riconoscere quali di queste parentele sono in particolare relazioni padre-figlio. Tuttavia è possibile utilizzare un metodo più diretto ricorrendo al numero deep. Alle condizioni sopra riportate basta aggiungere una condizione:

Per ogni $u, v \in U$, u è un padre di v se:

1. $post(u) > post(v)$
2. $leaf(u) \leq post(v)$
3. $deep(u) = deep(v) - 1$

3.1.3 Forma a stringa rappresentativa di un albero

La struttura di un documento XML si presta bene ad essere convertita in una stringa testuale nella quale le label di ogni nodo elemento o attributo vengono riportate inalterate e la struttura di annidamento dei nodi è espressa mediante l'utilizzo di parentesi.

Per aumentare la leggibilità di tale stringa si è scelto di ripetere la label del nodo sia all'inizio dell'esplorazione dei sui nodi interni, sia ad esplorazione conclusa.

Sempre allo scopo di aumentare la leggibilità si è scelto di ricorrere ad un separatore inteso come sequenza di caratteri *(-!-)*.

Riportiamo di seguito un esempio per mostrare come la struttura di un generico albero T può essere espressa anche tramite una stringa.

Supponiamo di avere l'albero T riportato nella figura 3.3 qui sotto.

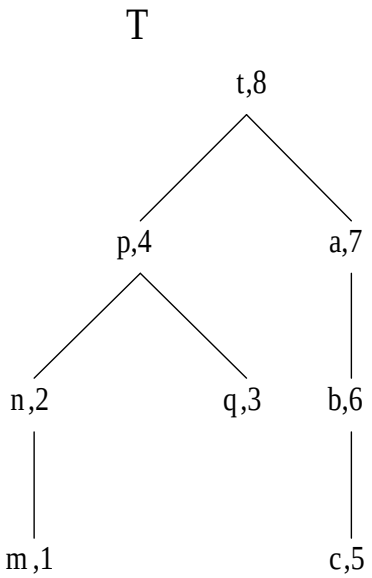


Figura 3.3: Esempio di albero

Ogni nodo è caratterizzato dalla propria label (etichetta) e ad ognuno è stato assegnato un numero seguendo la numerazione postorder. Questo albero T può essere espresso senza perdita di informazione tramite la forma a stringa:

```
(t(p(n(m-!-m)-!-n)(q-!-q)-!-p)(a(b(c-!-c)-!-b)-!-a)-!-t)
```

Una volta che ciascun nodo dell'albero è presente come oggetto indipendente è possibile esprimere senza perdita di informazione l'albero anche con la seguente stringa:

```
(8(4(2(1-!-1)-!-2)(3-!-3)-!-4)(7(6(5-!-5)-!-6)-!-7)-!-8)
```

dove ogni nodo non è più rappresentato dalla sua label ma dal suo numero in numerazione postorder. Poichè tale numero è unico per ciascun nodo, possiamo affermare che si comporta come una chiave per il nodo cui si riferisce e può quindi essere utilizzato in sostituzione al nodo stesso.

Una prima osservazione è quella che in questo modo si appesantisce inutilmente la rappresentazione della forma a stringa in quanto si inserisce la ripetizione delle label dei nodi foglia senza che da questa ne scaturisca un miglioramento nell'interpretazione.

Tuttavia rendere la rappresentazione uniforme si è poi deciso di mantenere tale metodologia di inserimento dei nodi anche per l'inserimento nella forma a stringa delle foglie.

Per convertire il documento XML nella stringa testuale rappresentante la struttura ed il contenuto dell'albero, il primo passo è stato creare un foglio di stile XSLT le cui regole trasformano un documento XML nella forma a stringa da noi voluta come ingresso.

Nel foglio di stile, sfruttando il lingauggio X-PATH descritto al nel Capitolo 1 sono inserite le istruzioni per riconoscere ed esplorare la struttura del documento XML distinguendo ogni nodo elemento da tutti nodi testuali. Per ogni nodo elemento riconosciuto si procede alla sua esplorazione per ricercare tutti i nodi in esso contenuti. Durante l'esplorazione il valore degli elementi o degli attributi viene inserito nel documento in uscita come label.

L'operazione di parsing del documento XML in ingresso è affidata al parser JDOM dalle librerie java, che ricevendo come ingresso il documento XML e il foglio di stile XSLT si incarica poi di trasformare il documento nella forma a stringa desiderata.

La forma a stringa ottenuta, espressione della struttura ad albero del documento XML originale, può ora essere presa in considerazione e convertita in un oggetto (nella realizzazione in java) albero utilizzato dall'algoritmo per il calcolo della tree edit distance.

Tuttavia la forma a stringa adottatta per rappresentare l'albero insito nella struttura del documento XML traduce solo le informazioni che sono contenute nel documento.

Ciascun nodo dell'oggetto albero che l'algoritmo elabora è costruito per contenere molte più informazioni.

Ricordando la definizione di tree edit distance osserviamo che a ciascun nodo dell'albero può essere assegnato un costo di cancellazione diverso da quello degli altri. A ciascun nodo può quindi essere assegnato un costo di cancellazione. Questa informazione non è espressa nel documento XML e quindi la forma a stringa non la contiene.

È però necessario che le informazioni necessarie all'algoritmo siano inserite correttamente in ciascun oggetto nodo. Ad esempio, per assegnare a ciascun nodo il proprio costo di cancellazione sono possibili diverse soluzioni:

- Nella conversione da forma a stringa ad albero, a ogni nodo viene attribuito lo stesso costo di cancellazione definito prima della conversione. In tale soluzione si suppone che tutti i nodi abbiano la stessa importanza. Sebbene questa soluzione possa essere bene accolta per il calcolo della tree edit distance tra alberi, risulta del tutto inadeguata nella rappresentazione dell'albero query. L'informazione contenuta nelle foglie dell'albero query è in generale più importante di quella contenuta nei nodi intermedi.

- Per ogni nodo analizzato si richiede il valore del costo di cancellazione.

Nel calcolo della *dissimilianza* tra due alberi T e T' ciascuna cancellazione di un nodo in T (inserimento in T') e ciascuna cancellazione di un nodo in T' (inserimento in T) sono operazioni che contribuiscono all'aumento della distanza. Il problema di risoluzione di una query però, si scosta da questo. La risoluzione di una query può essere ricondotto, come esposto al capitolo 2 all' approximate path inclusion problem. Infatti a noi non interessa determinare se l'albero query è simile e quanto è dissimile dall'albero dati ma, se questo contiene in alcune sue parti, sottoalberi simili all'albero query e determinare la relativa dissimilianza.

Così, mentre ha senso assegnare un costo alla cancellazione di un nodo nell'albero query, risulta del tutto inappropriato attribuire un costo alla cancellazione ai nodi dell'albero dati. Esistono solo alcune eccezioni alla considerazione fatta e saranno prese in considerazione sucessivamente quando parleremo di VLDC (Variable Length Don't Cares).

Nel confronto permetteremo quindi all'albero query di essere rappresentato in una o più parti dell'albero dati. A questo scopo è quindi possibile potare i sottoalberi dell'albero dati a costo zero.

Nel calcolo della distanza tra due alberi intesa come risoluzione di una query assegneremo quindi a ciascun nodo dell'albero dati un costo di cancellazione nullo.

3.1.4 Premesse all'algoritmo realizzato

Come detto, il nostro lavoro posa le fondamenta sull'algoritmo proposto da Shasha in [13]. La complessità di tale algoritmo è di tipo NP-completo e tale rimarrà anche nella nostra rielaborazione. Tuttavia, il tipo di approccio da noi seguito si scosta da quello proposto da Shasha in [13]. In particolare, in [13], per ogni marking $(S_T, S_{T'})$ considerato si calcola la distanza tra i rispettivi sottoalberi ridotti $RK(T)$ e $RK(T')$. Tenendo conto del fatto che i marking saranno considerati come stati, per determinare nello spazio degli stati quelli a cui corrisponde un minimo locale deve necessariamente determinare subito il costo del marking costruito.

La nostra idea è quella di effettuare una ricerca esaustiva dei marking e di conseguenza di tutti i marked tree che è possibile ottenere da ciascuno dei due alberi. Dati due alberi T e T' possiamo quindi considerare separatamente il problema di determinare i marked tree di T da quello per determinare i marked tree di T'. Questo metodo di procedere richiede senz'altro una complessità in termini di spazio in memoria superiore a quella dell'algoritmo in [13] ma riduce la complessità computazionale in termini di tempo per la costruzione esaustiva dei marking o rispettivamente dei marked tree.

La nostra idea nasce da due osservazioni che è possibile fare a riguardo dei marked tree che, una volta costruiti devono essere confrontati, misurando la distanza che esiste fra loro.

Teorema 3.1 *Per ogni Marking K tale che:*

$$Head(RK(T)) \neq Head(RK(T'))$$

il costo di tale marking K è infinito

Dimostrazione:

Supponiamo che $Head(RK(T)) > Head(RK(T'))$ *allora esiste un livello* l *in cui esiste un nodo* $i \in RK(T)$ *tale per cui:*

$$deg(RK(T)[i] \neq deg(RK(T')[j] qualunque j \in RK(T')$$

Quindi $RK\_tdist(root(T), root(T')) = \infty$

Teorema 3.2 *Sia liv(T) il numero di livelli dell'albero T. Possiamo affermare che dati due alberi T e T':*

$$liv(RK(T)) \neq liv(RK(T')) \Rightarrow cost(K) = \infty$$

Il primo passo da effetture per la riduzione della complessità computazionale del metodo scelto per il calcolo della tree edit distance è quello di ridurre gli alberi in ingresso. Sia l'albero dati che l'albero query vengono ridotti e cioè si creano due nuovi alberi nei quali compaiono solo nodi head. L'algoritmo sotto descritto prenderà in considerazione tali nuovi alberi i quali contengono potenzialmente un minor numero di nodi.

3.2 Marking

3.2.1 Costruzione dei marking

Nell'algoritmo da noi realizzato i due alberi vengono inizialmente considerati distintamente, cioè per ciascuno dei due alberi T e T' sono generati tutti i loro possibili sottoinsiemi, rispettivamente S_T e $S_{T'}$ che realizzano tutti i marking K possibili tra i due alberi.

Senza perdere in significato ci riferiremo separatamente all'insieme S_T e all'insieme $S_{T'}$ definendo per ciascun albero l'operazione di marking.

Poichè nell'analisi che si desidera effettuare, per ciascun marking K non risultano importanti i sottoinsiemi S_T e $S_{T'}$ di nodi eliminati quanto invece gli insiemi dei nodi appartenenti rispettivamente agli alberi $K(T)$ e $K(T')$, la prima parte dell'algoritmo si occupa di ricercare e costruire tutte le combinazioni di nodi che possono formare gli alberi $K(T)$ e $K(T')$.

Senza perdere in significato quindi, con il termine marking, ci riferiremo non ad un possibile insieme di nodi S_T da elininare da T quanto al corrispondente reduced tree $K(T)$ ottenuto cancellando da T i nodi di S_T.

Definizione 3.1 ***Marking:***

Dato un albero T, definiamo marking di T indicandolo con K(T) uno dei possibili alberi radicati che è possibile ottenere da T attraverso una operazione di potatura.

Ogni marking $K(T)$ di T dovrà poi essere valutato in relazione ad ogni marking $K(T')$ di T' per calcolare nel complesso il costo delle due operazioni di marking.

Di seguito è riportata una coppia di marking possibili degli alberi $T1$ e $T2$ rappresentati precedentemente nelle figure 3.1 e 3.2.

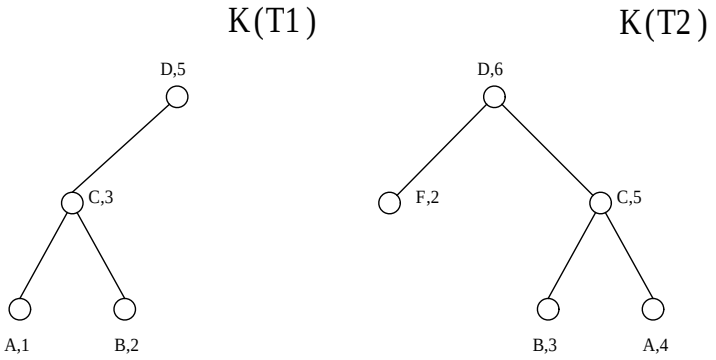


Figura 3.4: Esempio di marking non isomorfi

Si osserva subito che i due marking non sono isomorfi, pertanto il costo della coppia di marking in considerazione è infinito. Il risultato risulta ovvio in quanto il numero di nodi, nei due marking considerati è diverso. Non è pertanto possibile mettere in relazione ogni nodo di un marking con quelli dell'altro.

Nell'esempio seguente (figura: 3.5) i due marking sono isomorfi ed in particolare avremo che il miglior mapping completo esistente fra i due marking ha costo pari a 0.

La costruzione esaustiva di tutti i possibili marking rimane una operazione molto onerosa ma è l'unica che ci permette di determinare con precisione assoluta la tree edit distance fra alberi unordered. L'onerosità è dovuta all'elevato numero di combinazioni che possono esistere fra i nodi. Se tralasciamo per un attimo che ciascuna combianzione di nodi per essere un marking deve comunque essere un albero radicato, possiamo affermare che per un albero di n nodi, il numero delle combinazioni distinte che è possibile costruire con i sui nodi è $2^n - 1$

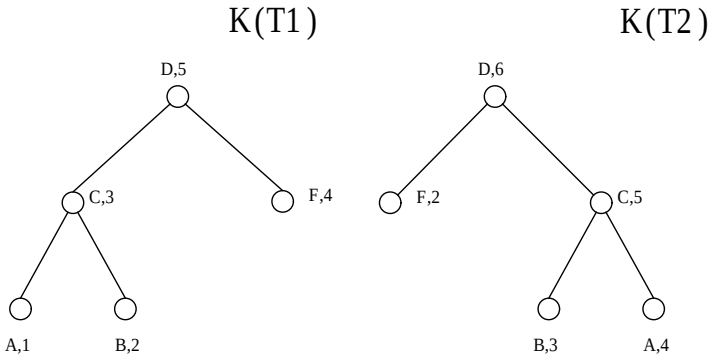


Figura 3.5: Esempio di marking isomorfi

Ricordiamo infatti che dato un insieme A la cui cardinalità è $card(A) = n$ allora le possibili combinazioni distinte tra i suoi elementi, cioè l'insieme dei suoi sottoinsiemi chiamato insieme delle parti è un insieme con cardinalità 2^n. Poichè a noi non interessa l'insieme vuoto, nel nostro caso le combinazioni possibili risultranno essere, come scritto sopra $2^n - 1$.

3.2.2 Algoritmo per la creazione dei marking

Nella progettazione dell'algoritmo si è cercato di limitare il più possibile la complessità computazionale.

Si è quindi sviluppato un algoritmo che genera tutte le combinazioni, che chiameremo sequenze, di nodi possibili considerandole come insiemi. Al primo passo vengono generate tutte le sequenze di nodi che ne contengono uno solo, il cui numero coinciderà col numero di nodi dell'albero ridotto di partenza. Al secondo vengono generate tutti le sequenze contenenti due nodi dell'albero e così via. L'algoritmo, al passo $i - esimo$, scorrendo tutte le sequenze generate al passo precedente e quelle contenenti un solo nodo, costruisce l'insieme che contiene tutte le sequenze con i nodi.

Nell'algoritmo utilizzato per generare queste combianzioni è necessario calcolare ad ogni $i - esimo$ passo tutte le combianzioni possibili contenenti i nodi anche se, alcune di esse non realizzeranno un marking. Infatti, una combianzione non marking rappresenta un albero non radicato, cioè un albero in cui manca una radice. Nel passo sucessivo, a tale combianzione può essere aggiunto un nodo che fungerà da radice mutando l'albero non radicato (sequenza di i nodi)

in un albero radicato (sequenza di $i+1$ nodi) che rappresenta quindi un marking per questo determinato albero .

Come detto, alcune di queste combianzioni, non saranno dei marking poichè i nodi contenuti non costituiranno un albero radicato. Il metodo per verificare se una sequenza di i nodi costituisce un albero radicato e quindi un marking sfrutta i numeri assegnati a ciascun nodo e risulta molto semplice.

Data la sequena di nodi S, sia $max(S)$ il nodo appartente alla sequenza con postorder maggiore allora, verificare che S è un albero radicato significa che:

Per ogni $u \in S$ $leaf(max(S)) \leq u \leq max(S)$

Per ogni combinazione generata, una volta verificato che la sequenza di nodi è un albero radicato, procedo con la riduzione di tale albero.

3.3 Riduzione degli alberi

3.3.1 Assunzioni per la riduzione dell'albero

Durante la riduzione dell'albero tutti i nodi non che non sono head vengono accorpati ad altri nodi.

È quindi stato necessario valutare le metodologie da utilizzare nell'accorpamento. Ogni nodo è definito, come illustrato precedentemente, da 5 valori, label, postorder, leaf, deep, del_cost. Abbiamo deciso che il nuovo nodo generato deve avere le seguenti caratteristiche:

- **Label:** La label del nodo accorpato è data dalla concatenazione di tutte le label dei nodi da accorpare. Ciascuna label viene distinta dalle altre attraverso un separatore che abbiamo scelto essere _.
- **postorder:** Il nodo accorpato mantiene il valore di numerazione postorder del nodo con postorder maggiore. In particolare mantiene quindi il postorder dell'antenato comune a tutti i nodi da accorpare.
- **Leaf:** Il nodo accorpato mantiene ancora il valore leaf del nodo con postorder maggiore.
- **deep:** Il valore deep del nodo accorpato è posto uguale al valore della deep del nodo sempre con postorder maggiore.

- **del_cost:** Il valore del_cost è calcolato come la somma di tutti i costi di cancellazione dei nodi da accorpare.

3.3.2 Risultati della riduzione dell'albero

Come risultato dell'operazione di riduzione oltre alla lista di nodi (head) richiedo l'espressione dell'albero in forma a stringa nella quale forma compaiono non più le label dei nodi ma i numeri di postorder assegnati a ciascun nodo.

Un esempio del tipo di informazione che si produce con la riduzione di un marking è riportato nella figura seguente (figura 3.6)

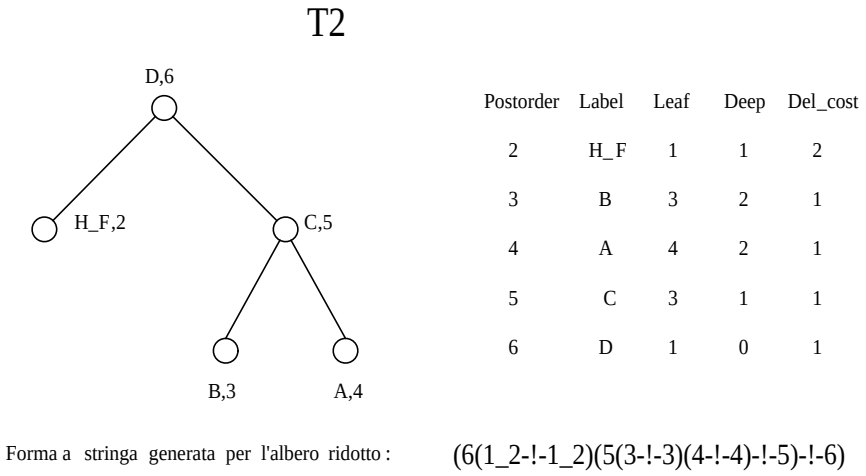


Figura 3.6: Esempio di albero ridotto: RK(T2)

Nella figura 3.6 si può inoltre osservare l'ordine con cui le label dei nodi sono concatenate e il fatto che nella forma a stringa generata per l'albero ridotto vengono concatenati anche i numeri in numerazione postorder dei nodi raggruppati.

L'identificazione dei nodi nella stringa tramite la numerazione postorder oltre a indicare precisamente qual'è il nodo o i nodi in caso di unione dovuta al processo di riduzione, dell'albero a cui ci si riferisce riduce l'occupazione in memoria di tale rappresentazione rispetto al caso si fossero inserite le label.

In questo modo ciascun albero ridotto sarà diverso dagli altri generati a partire dallo stesso albero.

3.3.3 Algoritmo per la riduzione dell'albero

Per ciascuno dei marking andiamo a determinare il corrispondente albero ridotto. L'algoritmo di questa operazione richiede che in ingresso l'albero sia nella solita forma. Deve essere espresso dalla lista di nodi ordinata secondo postorder crescente.

Scorrendo tale lista dal primo all'ultimo nodo, l'algoritmo identifica tutti i nodi selezionando quelli che sono head e quelli da accorpare. Riconosce poi globalmente la struttura dell'albero in modo da generare la forma a stringa rappresentativa dell'albero in esame.

L'algoritmo ricorre ad una struttura a stack di servizio nella quale vengono accatastati tutti i nodi per i quali non si sono ancora trovati tutti i fratelli ed una ulteriore lista a cui vengono appesi tutti i nodi non head. Una volta trovato il nodo head che è antenato di tutti i nodi della lista e padre dell'ultimo appeso alla lista stessa, tutta la lista viene inserita cone un unico nodo head.

È importante osservare che ogni volta che una nuova lista di nodi non head viene inizializzata, quella precedente e già stata svuotata e convertita in un nodo head. Questa proprietà è garantita dall'ordine in cui sono vistitati i nodi della lista ordinata.

Per ogni nodo n della lista esaminato viene preso in considerazione il nodo sucessivo u. Viene confrontato il loro valore $leaf$:

- Se $leaf(v) > leaf(u)$ allora il nodo u in esame ha sicuramente almeno un fratello quindi è un nodo head e può essere inserito tranquillamente nella forma a stringa così come è. In questo caso è anche chiaro che esistono suoi fratelli che non sono ancora stati presi in esame quindi inserisco tale nodo nello stack contente i nodi di cui non si sono trovati tutti i fratelli.

- Se $leaf(v) \leq leaf(u)$ allora il nodo u in esame è ultimo figlio del nodo v. Mi devo chiedere se nello stack ci sono nodi a cui manca almeno un fratello e se il fratello che manca è proprio u. Estraggo quindi il nodo w_0 in cima allo stack verifico:

  - $leaf(w_0) \geq leaf(v)$ allora w_0 è anch'esso figlio di v e quindi fratello di u. La prima considerazione che si può fare è che u è un nodo head e può essere inserito nella forma a stringa. Devo poi continuare ad estrarre dallo stack tutti nodi sino a quando questi sono figli di v e cioè tali per cui $leaf(w_i) \geq leaf(v)$

– $leaf(w) < leaf(u)$ allora nessuno dei nodi inseriti nello stack è suo fratello quindi u è unico figlio di v e non è quindi un nodo head. I nodi non head sono momentaneamente appesi in una lista temporanea in cui rimangono sino quando non si analizza il nodo padre dell'ultimo inserito nella lista che ha fratelli e che quindi rappresenta un nodo head.

Sia n il numero di nodi appartenenti all'albero da ridurre. La complessità computazionale dell'algoritmo è quindi n.

Una volta ottenuto l'albero ridotto lo andiamo a memorizzare in una struttura che contiene tutti gli $RK(T)$ fino ad ora generati. In tale struttura gli RK(T) sono mantenuti separati in base al numero di nodi ($HeadK(T)$) che contengono.

3.4 Calcolo della Tree Edit Distance

Una volta determinati tutti i sottoalberi ridotti sia per l'albero dati che per l'albero query possiamo iniziare a calcolare la tree edit distance tra i due alberi.

Ricordando la definizione 2.25 il calcolo della tree edit distance si riconduce quindi alla determinazione della coppia di markink $K(T)$ $K(T')$ rispettivamente per l'albero T e T' per la quale il costo è inferiore a quello di tutte le altre coppie possibili.

Come dimostrato nel teorema 3.1, possiamo limitare la complessità dell'algoritmo ed andare a determinare solo i costi delle coppie K(T) K(T') i cui rispettivi alberi ridotti RK(T) e RK(T') contengono lo stesso numero di nodi (uguale al numero di head).

3.4.1 Calcolo della distanza tra due nodi

Vogliamo ora soffermarci sulla funzione che calcola la distanza o dissimilianza tra due nodi. Come detto, gli alberi da noi considerati sono radicati ed etichettati, il che significa che ad ogni nodo è assegnata una etichetta e più precisamente, nel nostro contesto una stringa di caratteri rappresentativa del contenuto del nodo.

Calcolare la distanza tra due nodi significa allora determinare la distanza fra il loro contenuto e nel nostro caso determinare la distanza tra le loro etichette o label. Una funzione adeguata al contesto di risoluzione di una query, determinerebbe la distanza tra le label basandosi sul loro significato.

Ricordiamo l'esempio di query in figura 1.4 proposto in [1]. È evidente che per la query, nel calcolare la distanza tra il nodo query con label `book` e un nodo dati con label uguale a `document` occorre tenere conto del fatto che il significato dei due termini può essere nel contesto simile. Un algoritmo in grado di misurare tale similarità tra due label qualunque e restituire un risultato che sia una metrica è il primo strumento necesario per la risoluzioni di query approssimate.

La progettazione di tale algoritmo, tuttavia, esula dal contesto della tesi e non viene quindi presa in considerazione.

Questi tipi di algoritmi misurano la similarità tra due parole o frasi considerando ciascuno dei due termini di confronto non come sequenza di caratteri ma come oggetto con un proprio significato. La misura di similarità è dunque espressione di similarità nel significato degli oggetti medesimi. Tale metrica risulterebbe del tutto adeguata anche al problema di risoluzione di una query che stiamo analizzando.

Occorre però subito osservare che nessuno degli algoritmi esistenti può essere adottato in maniera diretta. Infatti, l'algoritmo da noi utilizzato per il calcolo della tree edit distance, sfrutta la riduzione degli alberi e crea nodi che sono l'unione di una sequenza di nodi originali. Nell'unione, come detto, le label dei nodi sono concatenate mantenendole comunque separate. Prima di applicare qualunque degli algoritmi citati sarebbe necessario decomporre le label composte per ridare significato a ciascuna singola label.

Per non appesantire ulteriormente il problema di risoluzione di query approssimate, nell'ambito del nostro lavoro abbiamo deciso di non adottare nessuna delle proposte citate ma di ricorrere al semplice calcolo della **edit distance** tra stringhe. In questo caso il confronto non tiene conto del significato e nell'unione di nodi e quindi di label non viene perso nulla.

Il confronto può continuare ad essere eseguito molto semplicemente anche per i nodi che non appartengono all'insieme dei nodi dell'albero query o a quello dell'albero dati, ma sono stati creati come unione di alcuni di questi che nel marking in esame non rappresentavano un nodo head.

La label di ciascun nodo è una stringa e viene da noi considerata come una semplice seuquenza di caratteri. In questo modo, nel nostro algoritmo, risulteranno simili nodi le cui label contengono caratteri uguali e nella giusta sequenza.

Per determinare la distanza tra due nodi abbiamo quindi deciso di calcolare la edit distance tra le rispettive label.

Tuttavia, per non generare ambiguità nell'algoritmo della tree edit distance nel quale esiste la possibilità di cancellare totalmente un nodo, è importante precisare che nel confrontare due label l_1 ed l_2 rispettivamente contenenti n_1 e n_2 caratteri sono contemplati diversi casi.

Sia $ed(l_1, l_2)$ la edit distnce tra le due label allora la distanza *dist* tra i due nodi vale a seconda dei casi:

- $dist = ed(l_1, l_2)$ se $ed(l_1, l_2) \leq n_1 \wedge ed(l_1, l_2) \leq n_2$
- $dist = \infty$ altrimenti

Se la distanza (edit distance) tra due stringhe è superiore alla lunghezza di quella minore significa che sono completamente diverse e che nessuna similarità (secondo il criterio da noi adottato) esiste tra i due nodi.

In questo caso l'algoritmo per il calcolo della tree edit distance non deve quindi prendere in considerazione il caso di *renaming* del nodo, ma quello di *cancellazione* del nodo query dalla query stessa. Assegnando un valore di infinito alla distanza, imponiamo all'algoritmo di non prendere in considerazione il caso di renaming o sostituzione del nodo ma quello di cancellazione.

3.4.2 Calcolo della distanza tra alberi ridotti

Come accennato precedentemente, ogni sottoalbero ridotto di T, $RK(T)$ viene memorizzato in una struttura dati capace di mantenere separati gli $RK(T)$ in base al numero di head che contengono. Analogamente, in una analoga struttura, sono memorizzati tutti gli $RK(T')$ di T'. Posso quindi scorrere parallelamente le strutture dei due alberi per confrontare e determinare i costi di tutte le coppie $(RK(T), RK(T'))$ dove il numero di head di $RK(T)$ coincide con quello di $RK(T')$.

Partiamo confrontando tutti gli $RK(T)$ con un nodo con gli $RK(T')$, anch'essi con un nodo calcolando le distanze tra loro che in questo caso coincidono con le distanze fra i singoli nodi head costituenti gli alberi ridotti.

Per ogni coppia $(RK(T), RK(T'))$, dopo aver calcolato la distanza tra i due sottoalberi, creiamo un elemento confronto contenente i riferimenti ai due alberi ridotti espressi dalla forma a stringa di questi e il valore della distanza fra loro.

Memorizziamo ciascun elemento confronto creato in una **tabella hash** che mi permetta in seguito un rapido accesso. Come campo per generare la chiave dell'elemento confronto nella tabella hash usiamo l'unione delle due forme a stringhe dei sottoalberi ridotti.

Per snellire il più possibile la tabella hash e quindi rendere più veloce l'accesso alla tabella stessa, abbiamo deciso di non memorizzare i confronti fra sottoalberi per i quali la distanza risulta infinito. In particolare, una volta confrontati i due alberi ridotti se la loro distanza risulta infinita allora l'algoritmo non inserisce nella tabella hash nessun nuovo elemento confronto.

Al passo sucessivo confrontiamo tutti gli $RK(T)$ con tre nodi (non esistono alberi rooted con due nodi) con gli $RK(T')$, anch'essi con tre nodi. Per determinare la distanza tra questi alberi ridotti dovremo solo andare a cercare le distanze fra i nodi già calcolate al passo precedente. Per prima cosa cercheremo nella tabella hash la distanza tra le radici dei due alberi, poi ricorrendo al grafo bipartito determineremo la migliore combinazione tra i figli.

Come più volte detto siamo interessati al calcolo della tree edit distance fra alberi unordered per i quali non è importante l'ordine da sinistra a destra che i figli di uno stesso nodo hanno fra di loro. Nell'algoritmo questa libertà comporta una ulteriore complessità. Occorrre isolare i figli e i sottoalberi di cui essi sono la radice nell'albero ridotto in considerazione. Dopodichè, avendo isolato tali sottoalberi anche per l'altro sottoalbero ridotto, occorre determinare il migliore complete mapping tra questi.

Nella figura 3.7 si osserva come per entrambi gli alberi ridotti in esame viene isolata la radice (nodo 3 per entrambi). Come detto viene ricercata la relativa distanza nella tabella hash. Poi, isolate le forme a stringhe dei sottoalberi radicati nei figli delle radici attraverso il grafo bipartito, si calcola il miglior mapping tra i sottoalberi. Nell'esempio i sottoalberi coincidono con un solo nodo, ma l'algoritmo, ricercando il confronto nella tabella hash, li considera come alberi radicati costituiti da un solo nodo.

In tale ricerca sono possibili diverse semplificazioni. La prima attuabile è verificare se il numero di figli di primo grado dei sottoalberi ridotti è diverso. In questo caso non sono isomorfi e tra loro esiste quindi una distanza infinita.

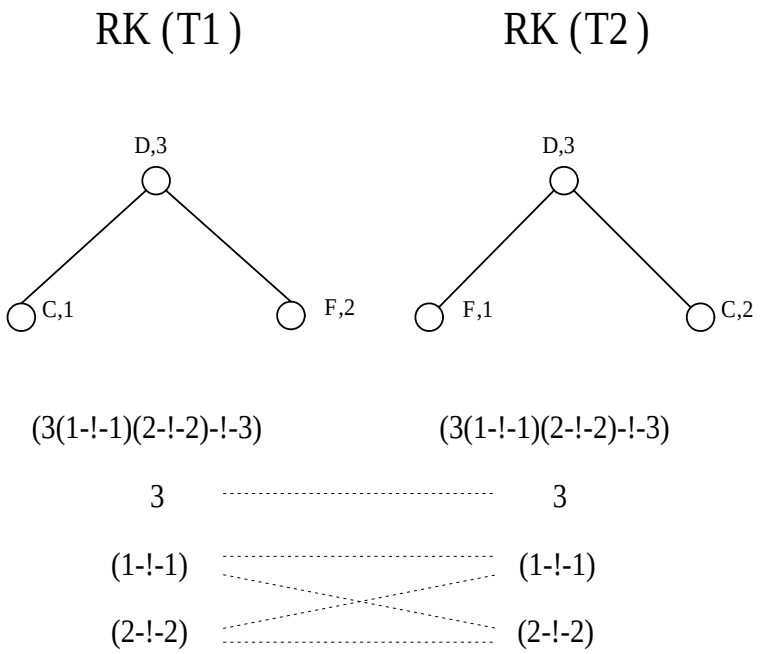


Figura 3.7: Identificazione dei sottoalberi tramite forma a stringa

Qualora il numero di figli sia uguale allora dobbiamo cercare il migliore mapping tra i figli e cioè tra i sottoalberi ridotti. In questa ricerca ricorreremo ai risultati ottenuti ai passi precedenti. Essendo i sottoalberi radicati dai figli, sottoalberi con un numero di head inferiore a quello dei sottoalberi $RK(T)$ e $RK(T')$ del passo attuale, tutti i possibili confronti saranno già stati effettuati ricordando che confronti non presenti comportano una distanza infinita.

3.5 Algoritmo ricorsivo per la ricerca del best mapping

Come precedentemente affermato, confrontando $RK(T)$ con $RK(T')$, dove entrambi gli alberi ridotti contengono lo stesso numero di nodi head, per prima cosa ricaviamo la distanza tra le due radici, poi occorre daterminare il miglior mapping tra i nodi figli (sottoalberi radicati nei nodi figli).

Il primo passo necessario è quindi l'identificazione dei figli per entrambi gli alberi. Tale ricerca risulta molto facilitata se si sfruttano le forme a stringhe.

Dalla forma a stringa di $RK(T)$ vengono isolate le forme a stringhe dei sottoalberi radicati dai suoi figli. Se il numero dei sottoalberi di $RK(T)$ è diverso

da quello di $RK(T')$, tra i due sottoalberi ridotti non può esistere un complete mapping, quindi la distanza tra loro sarà infinita e il costo del marking non viene salvato.

Se invece il numero di figli coincide occorre determinare qual'è il miglior mapping tra i figli.

Per ogni sottoalbero radicato in un figlio di $RK(T)[i]$ occorre cercare la sua rappresentazione tra i sottolaberi radicati nei figli di $RK(T')[j]$. L'insieme delle rappresentazioni scelte deve essere quella che minimizza il costo totale.

Occorre quindi cercare il mapping tra i sottoalberi radicati dai figli rispettivamente di $RK(T)[i]$ e $RK(T')[j]$ che complessivamente minimizza il costo del marking.

Nell'algoritmo tale operazione è facilitata poichè le distanze tra i sottoalberi di $RK(T)$ e di $RK(T')$ sono già state tutte calcolate mantenendo solo quelle che costituiscono un complete mapping.

Ricordiamo, infatti, che l'algoritmo considera un elemento di confronto non presente nella tabella hash come un confronto che aveva prodotto costo infinito.

Ricorrendo ad un algoritmo ricorsivo da noi progettato determiniamo quale rappresentazione in $RK(T')$ per ogni sottoalbero di $RK(T)$ è opportuno sceglere per minimizzare il costo del complete mapping.

Qualora tale rappresentazione esista, sommando i costi determiniamo la distanza tra $RK(T)$ e $RK(T')$. Per ciascun confronto nel quale la distanza non sia infinita, viene creato un *elemento confronto* e memorizzato nella tabella hash. Riportiamo di seguito l'esempio dell'elemento confronto creato dopo il confronto degli alberi riportati in figura 3.7.

Ogni elemento confronto è caratterizzato dalla propria chiave e oltre a contenere le forme a stringa dei due alberi ridotti confrontati contiene anche il valore della distanza prodotto dal miglior mapping tra i nodi dei due alberi e il mapping stesso. Mentre si ricerca il miglior mapping tra i sottoalberi questo viene anche memorizzato in una ulteriore forma a stringa per l'albero dati. Con tale rappresentazione si vuole mantenere traccia del miglior mapping trovato e permettere in seguito, scorrendo parallelamente la stringa dell'albero query con quella dell'albero dati riordinata, di ritrovare il miglior mapping possibile tra i nodi dei due alberi esaminati. Nella figura 3.8 osserviamo infatti che la forma a stringa

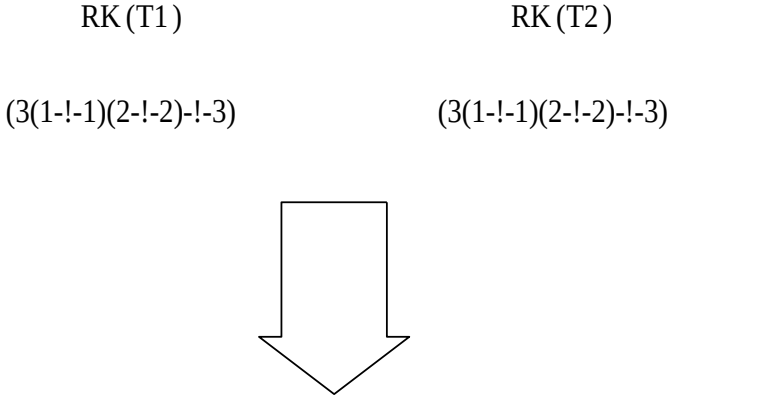


Figura 3.8: Elemento confronto

dell'albero ridotto ricavato dall'albero dati viene ripetuta invertendo l'ordine dei sottoalberi.

3.6 VLDC: variable length don't cares

Una volta realizzato l'algoritmo per determinare la distanza tra alberi occorre studiare come inserirlo efficacemente nel contesto di risoluzione di una query su dati semistrutturati come quelli XML. Nello specifico caso di risoluzione di una query non è richiesto di confrontare la struttura della query con tutta quella del documento, ma di ricercare nel documento le parti che risolvono la query e che quindi risultano ad essa simili rientrando in una certa soglia.

Il problema della tree edit distance può essere facilmente ricondotto a quello appena descritto se ad ogni nodo dell'albero dati è assegnato un costo nullo.

In questo modo è permesso, a costo zero, eliminare nodi dall'albero dati e ciascun marking costruito su questo ha quindi costo nullo. Possiamo considerare sottoinsiemi dell'intero documento contenenti un numero di nodi pari a quel-

lo della query e confrontarli con la query stessa senza che il confronto risulti penalizzato dall'aver potato anche parecchi nodi dall'albero dati.

Questa operazione risulta però troppo semplicistica. Siano u e v due nodi dell'albero query. Assegnare un costo nullo all'operazione di cancellazione di un qualunque nodo dall'albero dati significa anche svincolare la dissimilanza dai legami di parentela esistenti tra le rappresentazioni dei nodi della query nell'albero dati e permettere quindi l'esistenza di nodi nell'albero dati tra la rapprentazione di u e quella di v nell'albero dati senza incrementare in nessun modo la distanza tra la query e la sua rappresentazione nell'albero dati in esame.

Questo grado di liberta è noto col nome di *VLDC* e permette l'esistenza di nodi nell'albero dati tra la rappresentazione del nodo u e quella del padre di u a costo 0.

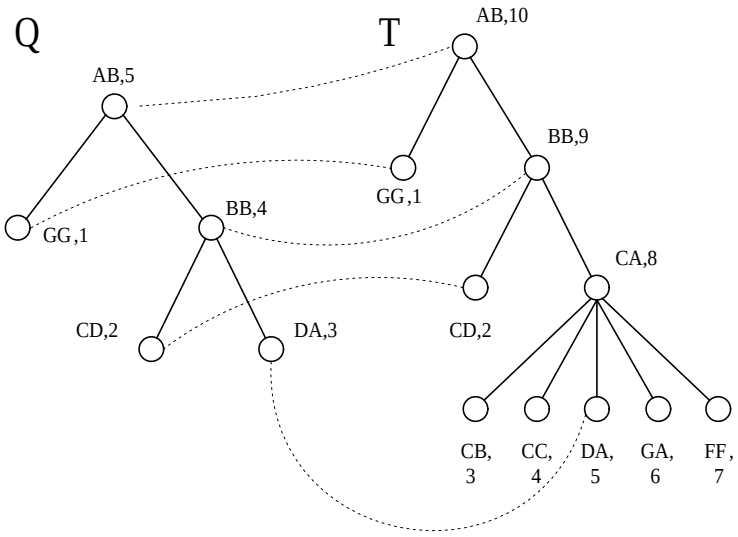


Figura 3.9: Esempio di VLDC

Nella figura 3.9 viene mostrato un esempio di mapping tra i nodi della query Q e un sottoinsieme di quelli di T. Il mapping ha un costo nullo poichè tutte le label sono preservate. Assumendo che per tutti i nodi di T il costo di cancellazione sia nullo, anche il costo del marking che considera solo i nodi presi in considerazione dal mapping è nullo. In questo caso troveremmo quindi che la query Q è risolta nell'albero dati T con distanza 0. Tuttavia, risulta evidente dalla figura che la struttura dell'albero dati non contiene perfettamente la struttura della query in quanto esiste il nodo dati `CA` tra i nodi `BB` e `DA` contenuti nel mapping.

Nella formulazione di una query, si possono volere esprimere non solo i contenuti ed i legami di parentela che devono esistere tra questi, ma anche quanto è possibile rilassare tali legami.

Se si vuole risolvere una query considerando VLDC per tutti i nodi della query, l'assunzione da noi fatta di assegnare costo di cancellazione uguale a zero per tutti i nodi dell'albero dati risponde a pieno alle esigenze del problema.

Tuttavia, l'utente nel formulare la query protrebbe voler distinguere nodi query per i quali è possibile la presenza di VLDC ed altri per i quali non lo è. Per questi ultimi potrebbe voler esprimere un eventuale costo sul rilassamento del vincolo di parentela padre-figlio ed eventualmente un massimo in termini di gradi di parentela al rilassamento che è possibile fare.

L'eventuale costo di questa libertà può essere espresso per ciascun nodo assegnandogli un costo per il rilassamento della relazione di parentela.

Occorre quindi aggiungere un altro numero alla caratterizzazione di ciascun nodo dell'albero che indichi il costo che comporta considerare come nodo padre un nodo che nell'albero dati risulta invece un antenato di grado maggiore. Con **cost_ril_par** indichiamo tale costo ed ogni estensione nel grado di parentela comporta un costo, quindi una distanza pari a $cost\_ril\_par$.

Sia u un nodo nell'albero query e v il padre per il quale vale la relazione: $deep(v) = deep(u) - 1$

Sia CM il complete mapping in esame, u' la rappresentazione nell'albero dati del nodo u e v' quella di v. Sia inoltre $cost(CM)$ il costo del CM nel caso in cui $v' = par(u')$ ed in particolare $deep(v') = deep(u') - 1$. Allora, qualora esista un CM' uguale a CM ma per il quale è w' la rappresentazione di v avremo che:

- se $deep(w') = deep(u') - 2$ allora $cost(CM') = cost(CM) + cost\_ril\_par$
- se $deep(w') = deep(u') - 3$ allora $cost(CM') = cost(CM) + 2 * cost\_ril\_par$
- se $deep(w') = deep(u') - a$ allora $cost(CM') = cost(CM) + a * cost\_ril\_par$

Volendo, come detto, esprimere un limite massimo al rilassamento riguardo le relazioni di parentela questo è assunto essere uguale per tutti i nodi e indipendente dal costo. Tale valore è chiamato **parentela** e vincola tutte le soluzioni alle query. Più precisamente possiamo affermare che dati u ,v appartenenti all'abero

query tali che $v = par(u)$, e dati u', v' rispettivamente le rappresentazioni nell'albero dati dei nodi u e v, allora anche qualora *cost_ril_par* per il nodo u valga zero, se il grado di parentela tra u' e v' è superiore a *parentela* questo mapping sarà scartato e non comparirà nelle soluzioni della query.

Capitolo 4

Realizzazione di un filtro basato sul contenuto

4.1 Limiti dell'algoritmo per il calcolo della tree edit distance

L'elevata complessità computazionale, di tipo NP-completo, del calcolo della tree edit distance fra alberi, come dimostarto da K. Zhang in [9], rende necessario l'utilizzo di filtri che limitino tale complessità.

Nel nostro ambito specifico di risoluzione di una query, la prima osservazione da effettuare è che la dimensione della query, cioè il numero dei nodi che la compongono, è sempre limitato.

Negli esempi di query riportati in precedenza i nodi che la compongono sono 6 nella figura 1.2 e sempre 6 nella figura 2.3. Riportiamo ora un ulteriore esempio di possibile query proposto da P. Ciaccia e W. Penzo in [4]

Esempio 2 *Ricercare tutti i dati dei cd musicali in vendita negli stores di Manhattan per i quali, il nome dell'autore sia Jhon, l'anno di produzione il 1996 e contengano una canzone lyric.*

Tale interrogazione verbale viene tradotta formalmente nel seguente modo:

```
cdstore/cd[author/lastname = "Jhon" and
           ../address/* = "Manhattan" and
           @year = "1996" and .//song/lyric]
```

L'esempio di query riportato nell'esempio precedente può essere tradotto nella seguente struttura ad albero.

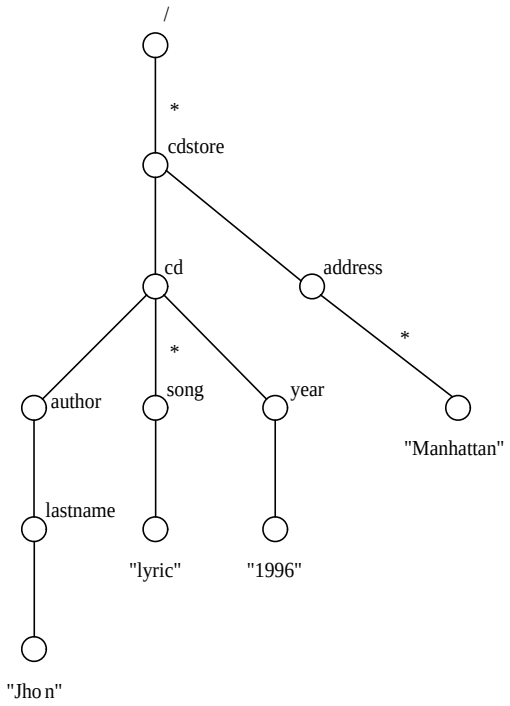


Figura 4.1: Esempio di query

In questo esempio il numero di nodi che la costituisce è dodici. Possiamo quindi concludere che tanto nella realtà quanto negli esempi proposti negli articoli dai vari autori, il numero dei nodi che compongono una query sia mediamente dieci.

Viceversa, il numero di nodi che compone l'albero dati può essere arbitrariamente grande.

Il file *dblp* [15] in formato XML da noi utilizzato per effettuare i test ha una dimensione di 138 Mbyte. Tale archivio, che noi abbiamo utilizzato solo in parte, è certamente costituito da diverse migliaia di milioni di nodi. Come detto, la trattazione di un tale archivio non è stata presa in considerazione nell'ambito del nostro lavoro poichè avrebbe richiesto all'algoritmo di lavorare in memoria secondaria.

L'obiettivo dell'algoritmo rimane quello di trovare le soluzioni accettabili ad una query che interroga un albero le cui dimensioni rimangono comunque limitate

dalla disponibilità di memoria principale. L'algoritmo ha quindi la pretesa di risolvere la query rispetto ad alberi relativamente grandi che noi consideremo essere composti di 500 nodi.

Occorre subito osservare che, l'algoritmo proposto al capitolo precedente richiede una complessità computazionale che è esponenziale rispetto al numero di nodi dei due alberi. È pertanto impossibile applicare tale algoritmo anche ad alberi relativamente grandi. Dai test effettuati risultano già inaccetabili, come tempi, ma soprattutto come occupazione di memoria, alberi con 18 nodi head.

L'algoritmo risulta quindi incapace di calcolare la distanza tra due alberi con numero di nodi superiore a 17 e quindi incapace di risolvere la query. Deve perciò, nella sua globalità, trarre vantaggio dalle osservazioni fatte e sfruttare il fatto che la dimensione delle query è sempre modesta.

La prima considerazione che occorre ricordare è che noi siamo interessati a ricercare le parti dell'albero dati che sono simili all'albero query e che quindi risolvono la query stessa, mentre non ci interessa affatto il confronto di tutto l'albero query con la totalità dell'albero dati.

Risolvendo la query ci chiediamo se esistono sottoparti dell'albero dati che risultano, a meno della soglia massima fissata, simili all'albero query. Nella figura 4.2 evidenziamo questo fatto riconoscendo in Q' una sottoparte di T che è simile alla query Q. Andando poi a determinare, utilizzando l'algoritmo per calcolare la tree edit distance discusso al Capitolo 3, la distanza tra gli alberi Q e Q' otteniamo che la query e risolta con distanza 1, pari al costo di renaming del nodo `CA` nel nodo `BA`.

Il progetto da noi realizzato, per limitare la complessità insita nel modello scelto per la risoluzione della query, ricorre ad una preelaborazione del documento su cui effettuare la query, volta a ricercare nel documento dati le parti che possono soddisfare la query.

Nel processo di preelaborazione abbiamo cercato di inserire i criteri di selezione e confronto meno sfruttati dall'algoritmo per il calcolo della tree edit distance. Abbiamo quindi cercato i difetti del metodo scelto per predisporre il processo di preelaborazione a limitare gli aspetti meno efficienti dell'algoritmo presentato al Capitolo 3.

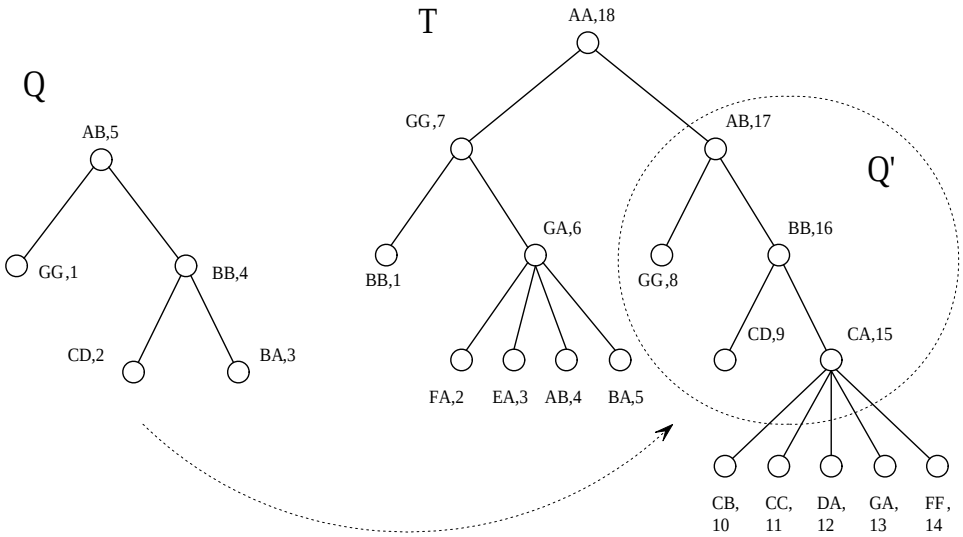


Figura 4.2: Ricerca delle sottoparti dell'albero dati T

La prima critica da rivolgere al metodo scelto è quella che trascura inizialmente del tutto il contenuto dei nodi. Infatti, ricercando per l'albero query e quello dati i reduced tree, ignora completamente quali dovranno essere i valori delle label dei nodi. A dimostrazione di questo basta ricordare che i reduced tree sono costruiti prima per l'albero query ignorando quello dati poi per l'albero query ignorando quelli già determinati per l'albero query. Nel calcolo della tree edit distance, solo alla fine, quando viene calcolata la distanza tra due reduced tree si calcola l'effettiva distanza nel contenuto fra i nodi.

Inoltre, tale modello, è adatto per il calcolo della distanza tra alberi le cui dimensioni sono pressochè uguali. Infatti anche limitando la creazione dei reduced tree per l'albero dati ad alberi che hanno un numero di nodi non superiore al numero di nodi della query (quindi minore), il numero dei reduced tree da considerare rimane alto poichè si limita a troncare la serie:

$$\sum_{i=0}^{n} C_{n,i}$$

limitandola a:

$$\sum_{i=0}^{m} C_{n,i}$$

dove i numeri n è la dimensione dell'albero dati m la dimensione dell'albero query e $C_{n,i}$ sono detti *coefficienti binomiali* e rappresentano il valore:

$$C_{n,i} = \frac{n!}{i!(n-i)!}$$

È allora necessario che le combianzioni di nodi per la creazione dei marking sull'albero dati non prendano in considerazione l'intero insieme dei nodi di T ma solo il sottoinsieme dei nodi Q' che può risolvere la query Q (figura 4.2).

Esempio 3 *Sia $n = 15$ il numero di nodi di T e $m = 5$ il numero di nodi di Q.*

Il numero di possibili combinazioni di nodi di T o cardinalità dell' insieme della parti di T è dato da:

$$2^n - 1 = 32768 - 1 = 32767$$

Qualora decidessimo di considerare solo le combinazioni con un numero di nodi non superiore ad m otterremmo un numero di combinazioni pari a:

$$15 + 105 + 455 + 1365 + 3003 = 4943$$

Mentre il numero di combinazioni possibili per un insieme di 5 nodi (come Q o Q') è dato da:

$$2^m - 1 = 32 - 1 = 31$$

Dall'esempio si osserva sibuto la riduzione della complessità ottenuta se si considera solo il sottoinsieme dei nodi di Q', invece di troncare la serie rappresentante il numero di combinazioni possibili per T.

Una ulteriore osservzaione è che per la risoluzione di una query occorre determinare tutte le parti del documento (dell'albero dati) in cui compare un sottoalbero che è simile alla query (ricordando la già definita metrica di similarità). Non si tratta quindi propriamente del calcolo della tree edit distance.

Siano Q il nostro albero query e T il nostro albero dati. Siano $\mid Q \mid$ e $\mid T \mid$ rispettivamente il numero dei nodi contenuti in Q il numero di quelli contenuti in T.

Possiamo affermare che se, ipoteticamente, calcolassimo la tree edit distance tra Q e T, anche qualora esistesse T_1 sottoalbero di T esattamente uguale a Q la

distanza tra i due alberi sarebbe uguale al costo di cancellazione di tutti i nodi di T non appartenenti a T_1. Inoltre, se il costo di cancellazione per ogni nodo di T vale 1 la distanza tra i due alberi T e Q non sarà mai inferiore a $\mid Q \mid - \mid T \mid$

È allora necessario che il costo di cancellazione di un nodo che non appartiene al sottoalbero di T simile a Q venga considerato uguale a zero.

Il problema di risoluzione di una query deve quindi essere ricondotto al calcolo della tree edit distance tra Q e tutti i sottoalberi T_i di T che sono simili a Q.

Per ogni sottoalbero T_i, TDist(Q,T_i) calcola precisamente la distanza tra i due sottoalberi e quindi esprime la metrica di similarità voluta tra la soluzione trovata e la query.

4.2 Premesse alla ricerca basata sul contenuto

Occorre quindi preventivamente ricercare i sottoalberi di T che potenzialmente possono essere soluzione alla query. Come detto il calcolo della tree edit distance inizialmente ricerca le soluzioni solo sulla base della struttura.

Nel progettare l'algoritmo per determinare i sottoalberi T_i è quindi opportuno che la *ricerca sia fatta sulla base del contenuto* espresso dai nodi dell'albero dati e dell'albero query.

Per migliorare l'efficienza è possibile inserire nella preelaborazione il controllo sulla massima distanza, in termini di parentela, che può esistere nell'albero dati fra due nodi che sono la rappresentzaione di nodi della query in relazione di padre-figlio.

Se questo controllo è totalmente eseguito nella preelaborazione la ricerca dei sottoalberi di T può spingersi oltre andando a selezionare di volta in volta gruppi di nodi che sono sottoalberi di T potati e non semplici sottoalberi. Non si tratta più solo di identificare T_i ma di selzionare i nodi di T_i che possono rappresentare i nodi della query potando quelli che non hanno nessuna corrispondenza nell'albero query. È opportuno osservare che per alcuni T_i possono essere possibili diverse potature e quindi un singolo T_i può generare più gruppi di nodi T_i' per i quali calcolare la tree edit distance con Q.

Consideriamo l'esempio della figura seguente (figura 4.3).

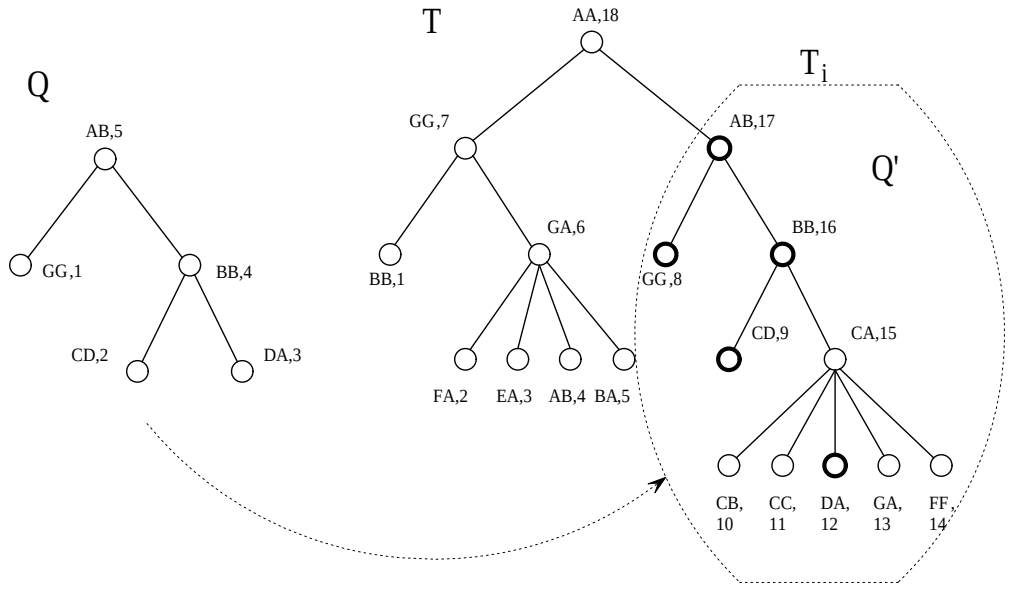


Figura 4.3: Ricerca delle sottoparti dell'albero dati T

Si osserva che il sottoalbero di T simile alla query risulta essere T_i ma che in T_i solo la combinazione di nodi evidenziati (Q') risulta del tutto simile alla combianazione di nodi di Q.

Ricordando che il numero di nodi di ciascuna sottoparte di T considerata determina in modo esponenziale il numero di combianzioni che esistono tra i nodi contenuti in essa, è conveniente isolare parti di nodi di T con numero di nodi al più uguale al numero di nodi della query. Naturalmente, in questo modo, diversi gruppi di nodi dovranno essere presi in considerazioni all'interno dello stesso sottoalbero T_i.

Ad esempio, in figura 4.4 viene mostrato che anche il gruppo Q'' deve essere confrontato con la query Q. La relativa distanza è pari a 1, ma rispetto all'esempio Q' in figura 4.3 tutti i legami di parentela contenuti in Q sono mantenuti inalterati.

In figura 4.3 la relazione padre-figlio fra i nodi `BB` e `DA` della query Q nel gruppo Q' è persa. Diremo che c'è stato un *rilassamento* sul vincolo di parentela.

Le parti isolate devono sempre e comunque sodisfare il vincolo di rappresentare un albero radicato come la query Q.

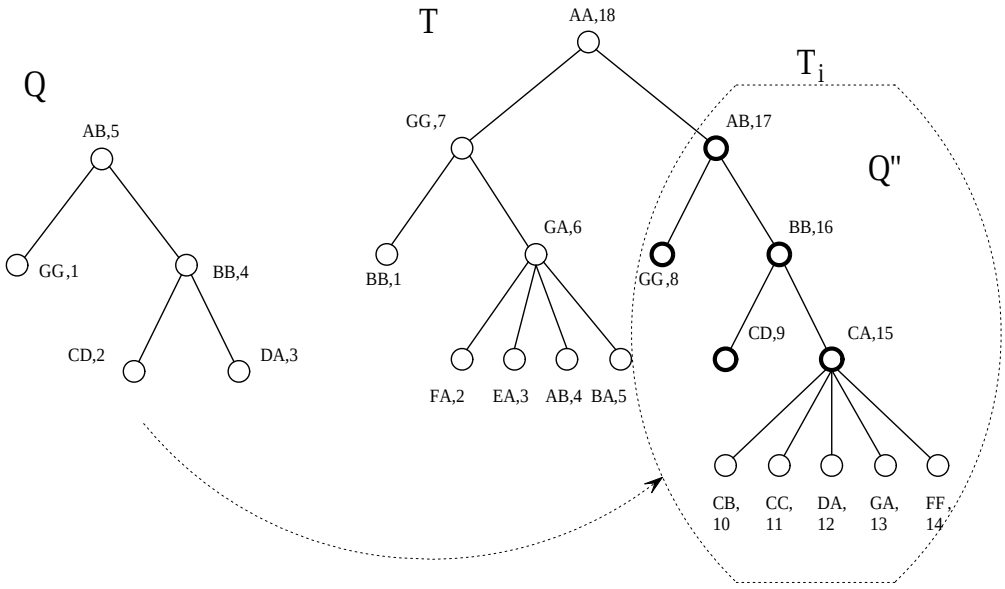


Figura 4.4: Ricerca delle sottoparti dell'albero dati T

4.2.1 Limiti alla dimensione della query

Procedendo come descritto al paragrafo precedente, scomponiamo il problema della ricerca dell'albero query nell'albero dati (tree inclusion problem) in una serie di problemi distinti di calcolo di similarità tra due alberi tra i quali il maggiore risulterà ora essere l'albero query.

Di conseguenza, per limitare la complessità dell'algoritmo per il calcolo dell tree edit distance è sufficente vincolare il numero dei nodi della query Q a non superare la soglia oltre la quale la complessità e l'occupazione di memoria diventano insostenibili rispettivamente per l'utente e l'elaboratore.

Dai test effettuati alla prima parte dell'algoritmo realizzato, descritto in precedenza al capitolo 3 si osserva che per un numero di nodi head dei due alberi da confrontare pari a 13, l'algoritmo lavora in tempi e con una occupazione di memoria accettabili.

Imponendo quindi un limite pari a 13, al numero di nodi head per la query, rendiamo il problema di risoluzione della query sempre possibile indipendentemente dalla dimensione dell'albero dati T. Ciò non è del tutto vero in quanto tutti i nodi dell'albero dati devono comunque essere mantenuti in memoria principale per permetere l'elaborazione.

Vorremmo subito osservare che imporre un limite al numero di head della query pari a 13 equivale in media, per un documento XML, ad un limite pari a 20 nodi.

Tale limite è, in questo ambito, del tutto accettabile poichè riteniamo importante in questo lavoro, non il confronto o la ricerca di sottoparti in documenti, ma la risoluzione di query.

4.3 Algoritmo per la ricerca basato sul contenuto

L'algoritmo per la ricerca basata sul contenuto è progettato per ricercare nell'albero dati i sottoinsiemi di nodi che hanno, a meno della struttura, una distanza inferiore alla soglia richiesta per soddifare la query.

L'idea è quella di considerare un nodo della query alla volta e verificare la sua presenza nell'albero dati. Il confronto non ricerca una ugualianza, ma restituisce la distanza tra il nodo della query e quello dell'albero dati che si stanno confrontando secondo la metrica di similarità esposta in precedenza al capitolo 3. In particolare, qualora tale distanza sia maggiore della soglia totale consentita alle risposte della query o risulti infinito, i nodi confrontati risultano completamente diversi.

Riportiamo in figura 4.5 il risultato della ricerca di tutti i nodi simili al nodo `AB`, radice della query Q in figura 4.3.

Si osserva che esistono sei nodi dati simili al nodo query AB e ognuno di loro deve essere considerato come possibile immagine della raidce di Q. Ovviamente solo quelli che ha senso considerare saranno considerati.

Il processo di esplorazione della query analizza i nodi secondo la loro profondità. Il primo nodo ad essere analizzato è la radice poi, uno ad uno i figli della radice finiti i quali si analizzano i nipoti e così via...

Per ogni nodo n della query analizzato vado a ricercare nell'albero dati tutti i nodi la cui distanza da n risulta diversa da infinito secodo i criteri definiti al Capitolo 3. In generale esistono più nodi simili al nodo n analizzato ognuno dei quali ha in particolare una specifica distanza da n e deve essere proposto come possibile immagine del nodo query n analizzato.

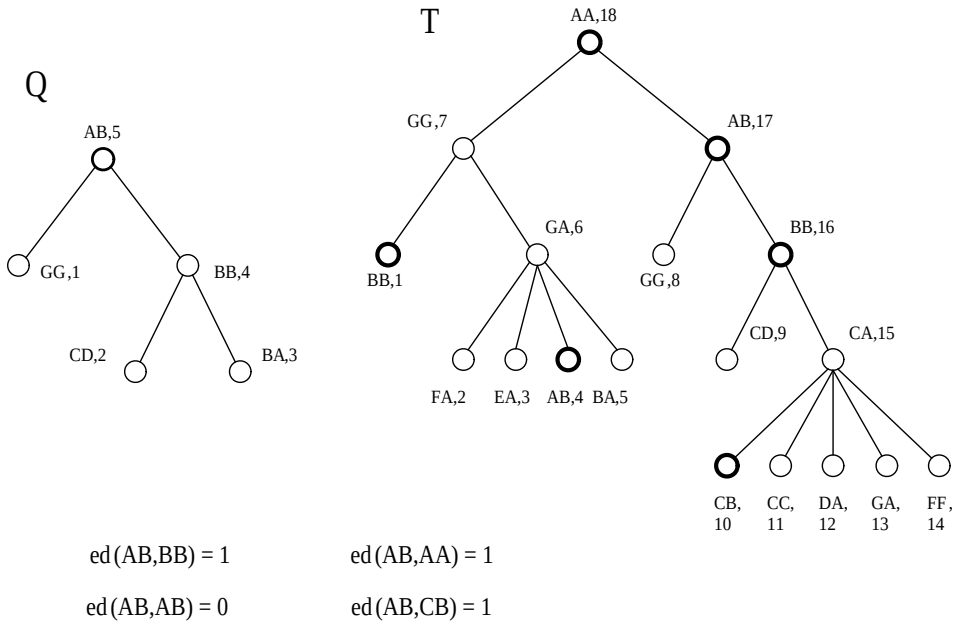


Figura 4.5: Ricerca dei nodi dell'albero dati T simili alla radice AB

È allora necessario che il filtro in progettazione sia in grado di riconoscere e creare tutte le possibili combinazioni di nodi dell'albero dati che possono soddifare la query avendo una distanza inferiore alla soglia imposta.

Per fare ciò, ogni nodo dati simile al nodo query radice viene proposto come possibile radice e viene inizializzata una nuova combinazione di nodi. L'esplorazione al passo sucessivo considera un figlio della radice dell'albero query. Tutti i nodi dati a lui simili devono essere proposti alle combinazioni inizializzate al passo precedente. Sucessivamente si cercano tutti i nodi dati simili ad un altro figlio della radice dati e si propongono tutti alle combinazioni sino ad ora generate. Qualora la radice non abbia altri figli si passa ad analizzare i nipoti cercando sempre i nodi dati a loro simili e procedendo come sopra.

L'idea è quella di considerare uno ad uno i nodi della query e cercare nell'albero dati quelli che possono rappresentarlo. Ogni $i-esimo$ nodo della query viene analizzato una ed una sola volta quindi al termine delle valutazioni riguardanti il nodo $i-esimo$, la distanza delle combinazioni deve essere ricolcolata secondo le operazioni che si è deciso di effettuare e come è stato modificato l'insieme delle combinazioni sopra citate.

Prima di dettagliare maggiormente l'algoritmo progettato per la realizzazione del filtro andiamo a definire alcune variabili:

- **costo_crescente**: è il costo di cancellazione di tutti i nodi della query sino ad ora processati.
- **del_cost** è il costo di cancellazione dell'$i-esimo$ nodo della query q.
- **dist** è il valore della distanza calcolata tra la label del nodo query q e quella del nodo dati d attualmente considerato e simile a q.

A ciascuna combinazione di nodi creata associamo poi un costo che indica la distanza che sino ad ora, scorrendo i nodi della query, ha accumulato dalla query stessa. Se questo costo è inferiore alla soglia, ha senso continuare a tenere in considerazione la sequenza, diversamente può essere gettata poichè è incapace di soddifare i requisiti della query ed è inutile aggiungervi gli altri nodi simili a quelli della query ancora da analizzare. Questo costo viene indicato con **costo_rel**.

Per prima cosa l'algoritmo analizza il nodo radice della query e ricerca nell'albero dati tutti i nodi simili. Dopodichè crea una combinazione di nodi per ogni nodo simile contenente proprio il nodo dati simile. Come detto ogni nodo simile trovato è contraddistinto da una distanza *dist* dal nodo query analizzato e, a tale distanza, viene inizializzato *costo_rel*.

Ad ogni nodo query analizzato viene incrementato il valore di *costo_crescente* del costo di cancellazione del nodo query. Inizialmente *costo_crescente* vale zero e quindi dopo aver processato il nodo query radice, vale il suo costo di cancellazione.

Poi passa ad analizzare un figlio della radice, incrementa il valore della variabile *costo_crescente* , e cerca nell'albero dati tutti i nodi a lui simili. Per ognuno di loro crea una nuova combinazione di nodi contenente il nodo stesso e per la quale il valore *costo_rel* vale $costo\_cresc + dist$. Naturalmente, ognuna delle possibili combinazioni è realmente inserita solo nel caso *costo_rel* sia minore alla soglia richiesta per le soluzioni alla query.

Questa operazione è necessaria poichè un nodo dati può essere simile a più nodi query e viveversa. È allora necessario permettere a qualunque nodo dati di rappresentare qualunque nodo query. Questa libertà aumenta notevolmente il numero delle combinazioni che è necessario generare ad ogni passo ma è l'unica che garantisce di non perdere eventuali soluzioni alla query.

Una volta inserite le combinazioni in cui la rappresentazione del nodo query figlio ne è la radice prova ad inserire ogni nodo simile alle combinazioni generate

al passo precedente. Una nuova combinazione viene creata per ogni inserimento di uno dei nodi simili in una delle combinazioni create al passo precedente. Non tutte le combinazioni sono però realmente create poichè non tutte rappresentano un albero radicato e queste vanno buttate. Inoltre le combinazioni che pure risultano alberi radicati devono soddisfare al requisito di possedere un *costo_rel* inferiore o uguale alla soglia. Per queste combinazioni *costo_rel* è calcolato come la somma di *costo_rel* della combinazione a cui si vuole aggiungere il nodo simile d_i piu la distanza *dist* tra il nodo query che si sta analizzando ed il nodo a lui simile d_i.

È inoltre necessario inserire ognuna delle combinazioni create al posso precedente aggiornando solo il valore *costo_rel* senza inserire alcun nodo. Anche questa operazione è necessaria per permettere a ciascun nodo dati di rappresentare qualunque nodo query. Supponiamo ad esempio che un nodo dati d risulti simile a due nodi query q_1 e q_2 che sono fratelli. Al nodo dati non sarebbe possibile rappresentare indistintamente un fratello o l'altro poichè una volta inserito in rappresentanza di un nodo non può ovviamente rappresentare anche un l'altro.

Per essere in grado di generare tutte le possibili combinazioni l'algoritmo ad ogni nodo query q analizzato deve quindi rigenerare tutte le combinazioni del paso precedente aggiornando *costo_rel* cioè incrementandolo del costo di cancellazione del nodo query analizzato al passo attuale. Come nei casi precedenti, le nuove combinazioni vengo realmente inserite solo qualora *costo_rel* risulti poi essere inferiore o uguale al valore della soglia.

In particolare per ogni nodo query q analizzato:

1. Scorre tutti i subtree fino ad ora generati. Per ognuno di loro:

   (a) Scorre e prova ad aggiungervi ognuno dei nodi simili. L'inserimento effettivo avviene solo nel caso

   $$costo\_rel + dist \leq soglia$$

   (b) Qualora $costo\_rel + costo\_del \leq soglia$ inserisce il subtree senza aggiungere alcun nodo. Questa operazione è necessaria per permettere ad un nodo dati simile a piu nodi query di rappresentare tutte le sue possibili corrispondenze.

2. Scorre tutti i nodi dati simili d_i al nodo query q in questione e se

$$costo\_crescente + dist \leq soglia$$

genera una nuova combinazione contenente al momento solo il nodo dati d_i che sarà la radice della combinazione.

Ad ogni inserimento nella lista di una combinazione che rappresenta un albero ottenuto da una libera potatura sull'albero dati, coincide il calcolo del valore *costo_rel* e tale inserimento avviene comunque solo se tale valore, per la combinazione costruita è inferiore o uguale alla soglia fissata nella query.

Ad ogni tentativo di inserimento di un nuovo nodo alle combinazioni esistenti, controlla che la nuova combianzione rappresenti un albero rooted, se così non è, la combinazione non viene aggiunta all'insieme delle combinazioni.

Questo garantisce che tutte le sequenze generate al passo attuale sono sottoalberi rooted e quindi rappresentano una possibile soluzione alla query.

Abbiamo quindi realizzato un filtro che crea le combinazioni di nodi che risultano nel contenuto simili ai nodi presenti nell'albero query e allo stesso tempo, ad ogni nodo della query analizzato, mantiene solo quelle che potenzialmente hanno una distanza inferiore alla soglia voluta.

Occorre poi precisare che con il metodo scelto per l'esplorazione dei nodi della query non si è in grado di mantenere le combinazioni di nodi create ordinate e gli alberi che alla fine si ottengono non sono quindi ordinati.

Alla fine della ricerca è allora necessario procedere con una operazione di ordinamento di tutti gli alberi trovati. Ricordiamo infatti che l'algoritmo del calcolo della distanza tra alberi lavora con rappresentazioni ordinate secondo postorder crescente.

Una volta identificati gli insiemi di nodi che rappresentano un albero radicato e che potenzialmente hanno una distanza inferiore alla soglia dall'albero query, l'obiettivo torna ad essere quello di calcolare la tree edit distance tra due alberi.

Ricordiamo infatti che, come affermato all'inizio della progettazione dell'intero lavoro svolto, noi riteniamo la tree edit distance l'unica metrica corretta per esprimere la distanza tra due alberi.

4.4 Criteri di selezione del filtro

Nella ricerca dei sottoinsiemi di nodi abbiamo però voluto aumentare il grado di selettività del filtro per ridurre il più possibile il numero dei sottoalberi dati da confrotare con quello query.

Come detto in precedenza lo scopo del filtro è ricercare i subtree sulla base del contenuto, cioè ricercare insiemi di nodi le cui label, globalmente, non hanno distanza dalle label della query superiore alla soglia. Abbiamo però cercato di includere nello stesso algoritmo anche ulteriori criteri di valutazione per l'eventuale inserimento di un nodo in una combinazione che tengano conto delle relazioni tra i nodi.

Senza aumentare troppo la complessità del filtro abbiamo inserito quindi anche controlli sulla struttura che i nodi possiedono fra loro. In questo modo alcune imperfezioni della struttura del gruppo di nodi dati selezionati, sono riconosciute e valutate durante la creazione delle combinazioni stesse. Valutandole al momento della creazione è possibile scartare di volta in volta quelle che non soddisfano ai requisiti imposti dal valore dei parametri .

Un primo controllo verifica che la relazione di parentela richiesta fra il nodo in esame e il padre trovato nella sequenza soddisfi i vincoli espressi nella query e aggiorni *costo_rel* qualora il possibile rilassamento sul vincolo non sia a costo zero.

Un ulteriore controllo per limitare i sottoinsiemi generati si basa sulla proprietà che se un sottoinsieme è contenuto in un altro allora la soluzione migliore, cioè la combinazione di nodi con distanza inferiore, si trova nel sottinsieme che contiene l'altro.

Ognuno di questi criteri di selezione ha lo scopo di ridurre quanto più possibile l'eventualità di calcolare la tree edit distance tra l'albero query ed alberi che non soddisfano la soglia massima accettata per le soluzione della query.

4.5 Creazione del dizionario

Ognuna delle verifiche sopra riportate è inserita allo scopo di velocizzare l'esecuzione di risoluzione della query.

Una ulteriore possibilità per velocizzare le interrogazioni su documenti XML è quella di sfruttare le caratteristiche che li contraddistinguono.

In un documento XML esiste una alta ripetizione di nodi intermedi (elementi). Un esempio pratico è l'archivio bibliografico dblp che contiene una grande quantità di riferimenti ad articoli, tesi, testi, manuali ed è quindi composto da diverse migliaia di milioni di nodi. È facile immaginare che per quasi la totalità dei documenti contenuti nell'archivio esiste un attributo *author* il cui valore è poi diverso a seconda del documento a cui si riferisce. Rimane, però il fatto che il nodo che nella rappresentazione ad albero del documento XML contiene la label *author* può comparire milioni di volte.

Risulta allora inutile confrontare un nodo query che possiede label *author* con tutti i nodi dati che possiedono label *author*. La prima osservazione è che una volta calcolata la distanza tra la label del nodo dell'albero query con quella di un nodo dell'albero dati, tale distanza è la stessa per tutti i confronti che riguardano lo stesso nodo query e un diverso nodo dell'albero dati ma che possiede un label identica. Nella ricerca di nodi dati simili ai nodi query risulta quindi del tutto inappropriato scorrere tutta la lista di nodi dati. Sarebbe più opportuno scorrere una lista dei nodi con label uniche in tale lista. La prima struttura che ci è sembrata in grado di risolvere questi problemi è quella classica di un **dizionario**.

Un dizionario italiano contiene tutte le parole della lingua italiana e per ognuna di loro riporta tutte le possibili definizioni. Questa struttura efficace nella ricerca dei significati delle parole italiane è la stessa che può essere adottata per rendere efficace, nel nostro algoritmo, la ricerca dei nodi dati simili ad un nodo query.

L'idea è creare un dizionario delle label contenute nell'albero dati dove ad ogni label (voce o parola) seguono tutte le definizioni e cioè i riferimenti ai nodi dell'albero dati in cui la label (voce) compare.

Per velocizzare quindi il confronto dei nodi della query coi nodi dell'albero dati, prima di ricercare le similarità, creiamo un dizionario. Una volta ottenuto il dizionario delle label dei nodi nell'albero dati, per ogni nodo query analizzato scorriamo tutte le voci del dizionario (che sono inferiori o al limite uguali al numero di nodi dell'albero dati) e calcoliamo la distanza tra ognuna di queste e la label del nodo query. Qualora il valore della distanza calcolata sia inferiore alla

soglia imposta alla query, preleviamo tutte le definizioni e inseriamo quindi nella lista dei nodi simili a quello query tutti quelli che compaiono come definizione.

Quando tutte le voci del dizionario sono state confrontate con la label del nodo query ho una lista contenente tutti i nodi simili a quello query.

4.6 Realizzazione del dizionario

Il dizionario è implementato come una classe java contenente una tabella hash di oggetti voci. Ciascun oggetto voce contiene la voce stessa che rappresenta e la lista delle definizioni che compaiono del documento XML per questa voce.

La creazione del dizionario avviene scorrendo tutti i nodi dell'albero dati. Ad ogni nodo viene prelevata la label che è una possibile nuova voce del dizionario.

Ogni voce con la sua definizione viene passata al dizionario che si occupa di inserirla appropriatamente. Come nuova voce con la sua definizione qualora non sia già presente una voce uguale, altrimente come nuova definizione aggiungendo il nodo attuale alla lista delle definizioni della voce uguale trovata.

La ricerca delle voci nel dizionario è molto efficiente poichè il dizionario è un tabella hash e l'acceso agli elementi avvine tramite codice hash.

Va osservato che un eventuale ordine delle voci del dizionario non comporta maggiore efficienza in quanto le similarità tra nodi non sono in alcun modo correlate.

Anche per l'albero query potrebbe essere creato un dizionario tuttavia data la sua dimensione limitata e la scarsa possibilità di presenza di nodi uguali, abbiamo deciso di non creare alcun dizionario per l'albero query.

Capitolo 5

Prove sperimentali e 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 dal programma da noi realizzato, sia per quanto riguarda gli algoritmi per il calcolo della tree edit distance, sia per la ricerca delle sottoparti dell'albero dati che possono soddisfare ai requisiti imposti dalla soglia della query.

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.

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 4 Gb.

5.1 Le collezioni usate nei test

Essendo la risoluzione di query su documenti XML 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. Nell'ambito della nostra

ricerca non esistono ancora collezzioni riconuscute e risulta ovvio che nessuna delle collezioni proposte per l'Information Retrieval può essere utilizzata.

Per effettuare le prove sperimentali sul sistema sono state pertanto utilizzate una serie di collezioni create ad hoc o proposte da altri ricercatori operanti nello stesso ambito:

- *Collezione "dblp"*: si tratta di un archivio bibliografico già citato in precedenza [15] e utilizzato anche in [1]. Ha una dimensione di 138 Mbyte e solo una parte di esso viene presa da noi in considerazione. Per la precisione, dell'intero archivio, vengono mantenuti solo i primi 1000 nodi (997).

- *Collezione "interna"*: si tratta di una collezione di diversi alberi costruiti ad hoc per verificare l'efficacia degli algoritmi progettati.

La collezione *interna* è stata utilizzata più che altro per analizzare il sistema qualitativamente, dal punto di vista dell'efficacia. L'idea alla base di questa collezione è quella di testare il sistema in una delle situazioni più comuni per il calcolo della tree edit distance. Per rendere più interessanti le prove, sono state scelte label per gli alberi, costituite da due caratteri. In questo modo è stato possibile distinguere l'ugualiaza dalla similarità tra label (nodi). Di conseguenza è stato possibile considerare casi in cui un nodo appartenente all'albero dati fosse simile a più nodi query. Tutti i nodi query della collezione interna hanno costo di cancellazione pari a 2. Essendo due i caratteri che compongono tutte le label dei nodi della collezione interna, abbiamo scelto tale valore per rendere uniforme il calcolo della distanza tra alberi.

La collezione *dblp* è invece stata prelevata dal sito internet [15] e ridotta per rendere possibile il suo utilizzo. Essa rappresenta un archivio bibliogarfico di notevoli dimensioni, pertanto, questa collezione è stata utilizzata soprattutto per testare il sistema nella sua globalità: sia dal punto di vista dell'efficienza che dell'efficacia; inoltre, è stata utile per verificarne il comportamento rispetto al contesto specifico di risoluzione di una query approssiamta su documenti XML, in cui il numero e la ripetitività degli elementi logici del documento XML stesso è molto elevato. Anche per tutti i nodi query della collezione dblp si è scelto un costo di cancellazione pari a 2.

Riportiamo di seguito, nell'esempio 4, una selezione dell'archivio dblp contenente 25 elementi più la radice `<dblp>` dell'intero documento.

Esempio 4

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="XMLTree.xsl"?>
<dblp>
  <article key="tr/gte/TR-0146-06-91-165">
    <ee>db/labs/gte/TR-0146-06-91-165.html</ee>
    <author>Alejandro P. Buchmann</author>
    <author>M. Tamer  Oumlzsu</author>
    <author>Dimitrios Georgakopoulos</author>
    <title>
       Towards a Transaction Management System for DOM.
    </title>
    <journal>GTE Laboratories Incorporated</journal>
    <volume>TR-0146-06-91-165</volume>
    <month>June</month>
    <year>1991</year>
    <url>db/labs/gte/index.html#TR-0146-06-91-165</url>
    <cdrom>GTE/BUCH91.pdf</cdrom>
  </article>
</dblp>
```

5.2 Risoluzione di una query approssimata

In questa sezione vengono analizzate le prestazioni del programma nella ricerca delle soluzioni ad una query approssimata. In particolare si esamineranno:

- l'*efficacia*, con la copertura raggiunta dalle risposte sulle collezioni e la qualità delle risposte trovate;
- l'*efficienza* dell'algoritmo nel suo comlpesso in varie modalità di utilizzo;
- gli effetti delle variazioni sui principali *parametri* disponibili che influenzano la selettività del *filtro* utilizzato.

5.2.1 Efficacia

Copertura

Un aspetto fondamentale dell'efficacia della ricerca di similarità tra alberi è la copertura che gli algoritmi di interrogazione sono in grado di fornire sulle varie collezioni. Per verificare tale copertura risulta più opportuno utilizzare la collezione *interna* poichè è stata appositamente creata per mettere in risalto tutte le complessità che possono esistere nel confrontare due alberi. Considereremo quindi come albero dati, l'albero T in figura 5.1 e come albero query, l'albero Q in figura 5.2.

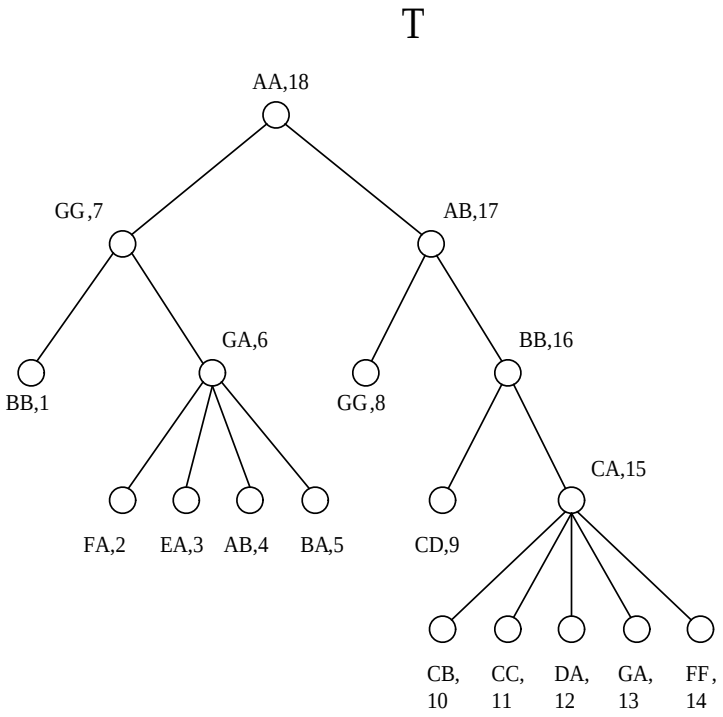


Figura 5.1: Albero dati T (*collezione interna*)

Il risultato dell'interrogazione riportato è relativo all'interrogazione Q (figura 5.2) eseguita sotto le seguenti condizioni:

- $soglia = 3$;
- $parentela = 2$;
- $ril\_par = 1$;

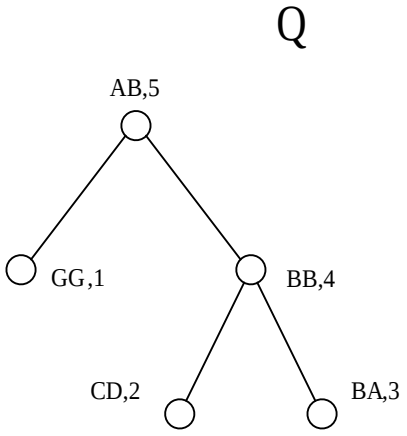


Figura 5.2: Albero query Q (*collezione interna*)

```
 Subtree selezionati    Soluzioni alla query Q con       distanza
 dal filtro:          distanza inferire alla soglia:

 17,8,16,9,13,   (17(8-!-8)(16(9-!-9)(13-!-13)-!-16)-!-17)  1+1
 17,8,16,9,12,   (17(8-!-8)(16(9-!-9)(12-!-12)-!-16)-!-17)  1+1
 17,8,16,9,15,   (17(8-!-8)(16(9-!-9)(15-!-15)-!-16)-!-17)  1
 17,8,16,11,15,
 17,8,16,10,15,
 17,8,16,15,13,
 17,8,16,15,12,
 18,7,16,9,1,
 18,7,16,9,5,
 18,7,16,9,6,
>18,7,16,9,15,   (18(7-!-7)(16(9-!-9)(15-!-15)-!-16)-!-18)  2+1
```

Nel risultato sono mostrati i gruppi di nodi selezionati dal filtro (prima colonna) e solo per quei gruppi la cui distanza è inferire alla soglia, sono riportate la forma a stringa (seconda colonna) e la relativa distanza con la query mantenendo separati il valore dovuto a dissimilianze sul contenuto da quello dovuto a dissimilianze sulla struttura.

Come si vede dai risultati ottenuti, per l'esempio di interrogazione Q sull'albero dati T, essi rappresentano tutte e sole le risposte alla query che rientrano nella soglia imposta.

Desideriamo ora far vedere come cambiano le soluzioni accettabili per la query variando i parametri dell'algoritmo. Il risultato dell'interrogazione sotto riportato è relativo sempre all'interrogazione Q in figura 5.2 ma eseguita sotto le seguenti condizioni:

- $soglia = 2$;
- $parentela = 2$;
- $ril\_par = 1$;

In questo test abbiamo scelto un valore di *soglia* minore, quindi ci aspettiamo che le soluzioni prima trovate che non soddisfano il nuovo valore della soglia non vengano riportate. Mostriamo ora i risultati ottenuti in questo test:

```
Subtree selezionati    Soluzioni alla query Q con       distanza
dal filtro:          distanza inferire alla soglia:

17,8,16,9,13,   (17(8-!-8)(16(9-!-9)(13-!-13)-!-16)-!-17)  1+1
17,8,16,9,12,   (17(8-!-8)(16(9-!-9)(12-!-12)-!-16)-!-17)  1+1
17,8,16,9,15,   (17(8-!-8)(16(9-!-9)(15-!-15)-!-16)-!-17)  1
17,8,16,15,13,
17,8,16,15,12,
```

Si osservi che, eseguendo l'interrogazione con il valore di soglia uguale a 2, nei risultati non si ottiene la soluzione evidenziata con il simbolo > nei risultati precedenti. Tale soluzione non era in grado di soddisfare il vincolo della soglia.

Vogliamo ora invece, far vedere come cambiano le soluzioni accettabili per la query variando il parametro *parentela*. Il risultato dell'interrogazione sotto riportato è relativo sempre all'interrogazione Q in figura 5.2 ma eseguita sotto le seguenti condizioni:

- $soglia = 3$;
- $parentela = 1$;
- $ril\_par = 1$;

Imponendo un valore di *parentela* inferiore ci aspettiamo che tutte le soluzioni trovate precedentemente che contenevano rilassamenti sul vincolo di parentela, non siano ora proposte come soluzione alla query. Riportiamo i risultati ottenuti in questo test:

```
Subtree selezionati    Soluzioni alla query Q con       distanza
dal filtro:          distanza inferire alla soglia:

17,8,16,9,15,   (17(8-!-8)(16(9-!-9)(15-!-15)-!-16)-!-17)  1
17,8,16,15,13,
17,8,16,15,12,
```

Osserviamo che esiste una solo gruppo di nodi in grado di soddisfare a pieno i requisiti imposti da *soglia* e *parentela*. Soprattutto il vincolo sulla parentela riduce molto i risultati accettabili. Ricordiamo infatti che imponendo il valore $parentela = 1$ imponiamo alle soluzioni della query Q nell'albero dati T di non presentare alcuna dissimilianza nei vincoli di parentela rispetto all'albero query Q stesso.

Osserviamo quindi che l'efficacia della copertura è totale e tutte le soluzioni trovata sono tutte e sole le soluzioni che soddifano la query rientrando nella soglia imposta. capace di limitare le combinazioni da considerare senza escluderne di corrette.

Vogliamo poi sottolineare come sia evidente l'efficacia del filtro realizzato: capace di limitare le combinazioni da considerare senza escluderne di corrette. In ognuno dei test effettuti il filtro non è in grado di selezionare solo i gruppi di nodi che costituiscono una soluzione (capacità non richiesta ad un filtro), ma si osserva subito che il numero di gruppi selezionati è in relazione al valore dei parametri.

Per la verifica dell'efficacia dell'algoritmo abbiamo utilizzato solo in parte la collezione *dblp* poichè non è in grado di esprimere tutte le particolarità ed il giusto grado di profondità per gli alberi. Nessuno dei test viene quindi riportato poichè appesantirebbe solo la lettura.

Nella collezione *interna*, esiste inoltre molta similitudine tra tutte le label dei nodi e esistono quindi molte combinazioni tra i nodi dati in grado di rappresentare, a meno della soglia, l'albero query. L'elaboriazione sulla collezione *interna* risulta pertanto anche più onerosa per l'algoritmo.

5.2.2 Efficienza

I documenti considerati

In questa sezione vengono mostrate le prestazioni relative all'algoritmo progettato. In particolare, vengono riportati i tempi delle varie query appartenenti alla collezione *dblp*, la più indicata per test di questo tipo date le sue ragguardevoli dimensioni.

Nelle figure 5.4 e 5.3 sono mostrati rispettivamente, una parte della rappresentazione ad albero del documento XML riportato precedentemente (esempio 4) ed un esempio di una albero query che è possibile rivolgere al documento. Entrambi appartengono alla collezione dblp sopra citata.

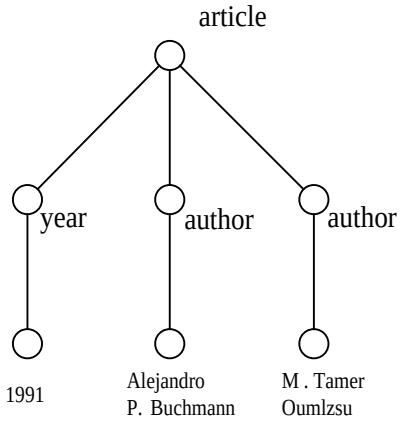


Figura 5.3: Esempio di query tree con 7 nodi (*collezione dblp*)

Abbiamo deciso di cosiderare questa collezione poichè è in grado di esprimere più verosimilmente le prestazioni dell'algoritmo essendo *dblp* un archivio bibliografico di documenti in formato XML presente sulla rete ([15]).

Per misurare l'efficienza dell'algoritmo realizzato abbiamo deciso di valutare due grandenzze:

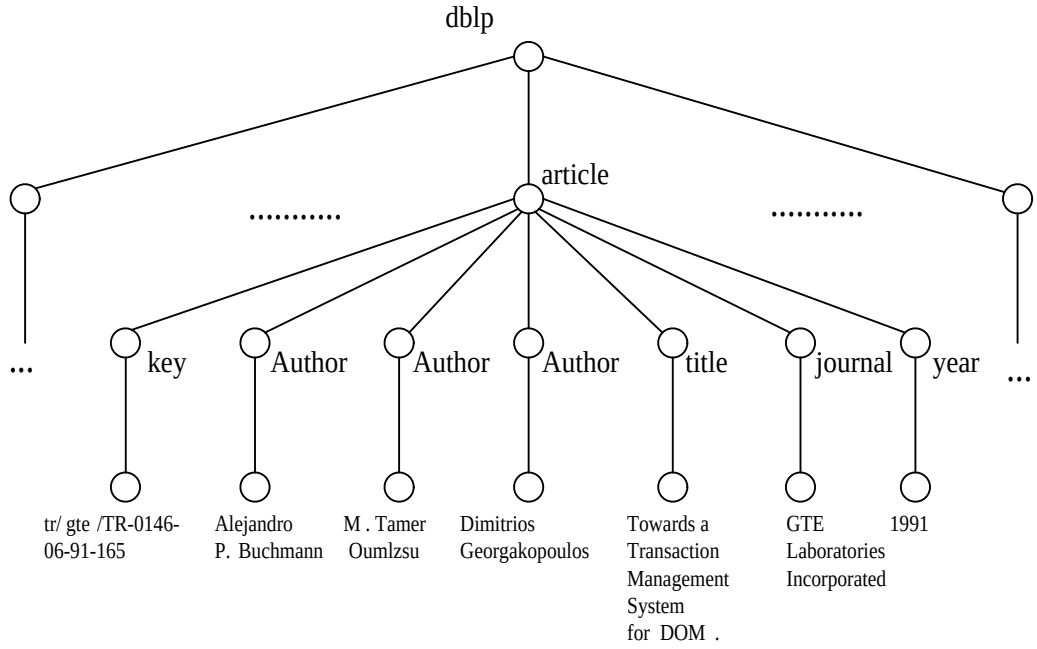


Figura 5.4: Esempio di data tree (*collezione dblp)*

- **Tempo di esecuzione:** durante l'esecuzione dell'algoritmo vengono presi tre tempi parziali espressi in secondi:

  1. t_{imm}: tempo parziale che indica il tempo necessario a convertire i dati in ingresso nella forma richiesta dall'algoritmo.
  2. t_{ric}: tempo parziale che indica dopo quanto tempo dall'inizio dell'esecuzione, l'algoritmo ha determinato tutti i gruppi dell'albero dati da considerare.
  3. t_{tot}: tempo totale impiegato dall'algoritmo per determinare tutte le soluzioni alla query analizzata.

- **Spazio di memoria occupata:** indica il picco massimo di memoria richiesta dall'algorimo per risolvere la query.

Occorre però precisare che, mentre il tempo di esecuzione viene misurato in modo preciso, l'occupazione di memoria viene valutata in maniera molto semplicistica e pertanto fornisce solo una indicazione sul comportamento dell'algoritmo nei test.

Come si vedrà, grazie sopratutto all'uso dei filtri mostrati nel Capitolo 4, i tempi per la risoluzione delle query riescono a mantenersi contenuti, soprattutto tenendo conto che la ricerca dei nodi simili riguarda alberi dati con una dimensione compresa tra i 350 e 1000 nodi.

Scalabilità

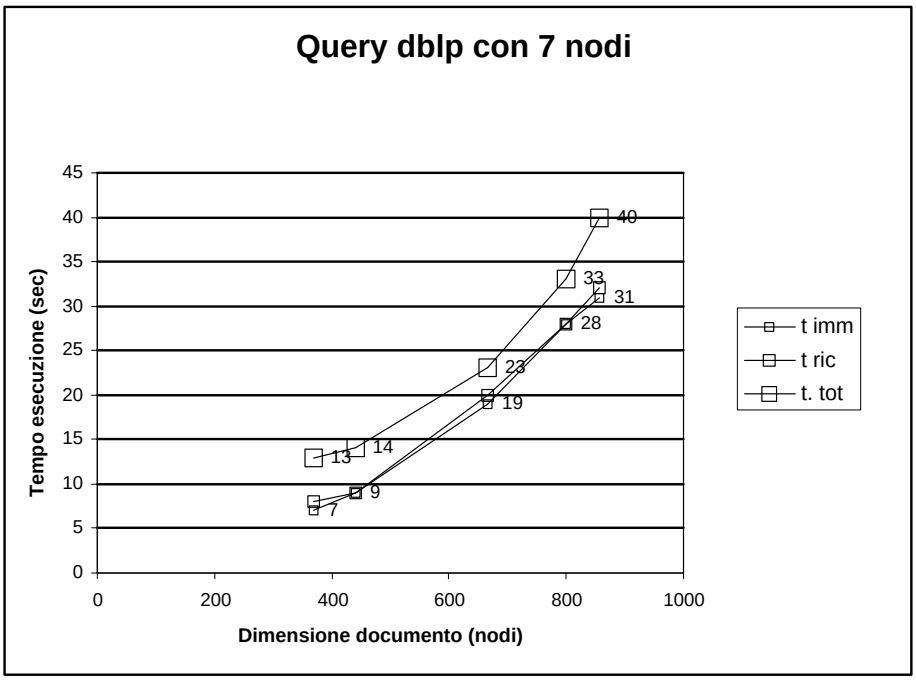


Figura 5.5: Test di scalabilità: tempi di esecuzione

Sulla collezione *dblp* sono stati quindi eseguiti i test di scalabilità distinti in due fasi. Nella prima fase di test, viene valutata l'efficienza al crescere del numero di nodi dell'albero dati e quindi la dimensione del documento su cui effetture la query. Sono stati quindi generati 5 file di documenti XML distinti e con una dimensione che varia dai 369 nodi a 857 su cui eseguire una stessa query (figura 5.2). Partendo col considerare il documento dati XML contenente 857 nodi, gli altri documenti utilizzati nei test sono stati ottenuti eliminando da questo via via un numero maggiore di elementi. Ogni documento contiene comunque la selezione di elementi riportata nell'esempio 4 da cui è ricavata la query.

I risultati per la prima fase di test, sono mostrati in figura 5.5 e 5.6 dove vengono riportate rispettivamente l'efficienza espressa in tempo di esecuzione e in memoria utilizzata.

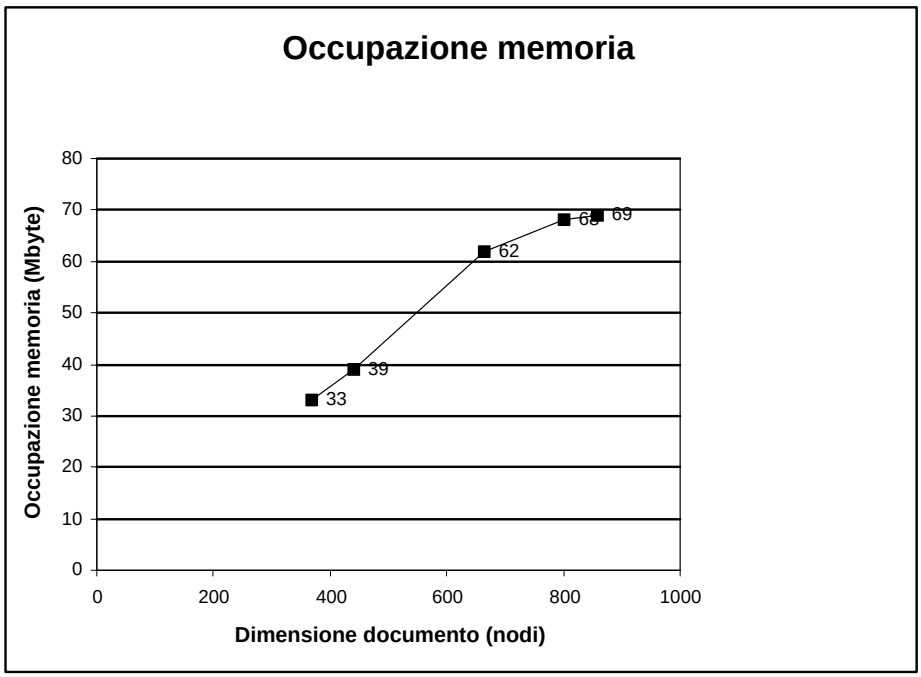


Figura 5.6: Test di scalabilità: occupazione memoria

Come messo in evidenza dai test:

- i tempi di risoluzione della query presentano un andamento più che lineare all'aumentare del numero di nodi del documento XML considerato. Questo è dovuto al fatto che il numero dei gruppi di nodi dell'albero dati che possono soddifare la soglia della query cresce in modo più che lineare col numero di nodi dell'albero dati stesso;

- l'occupazione di memoria presenta un andamento lineare all'aumentare del numero di nodi del documento dati XML. Infatti, l'occupazione di memoria dovuta all'albero query è costante, mentre siccome tutti i nodi dell'albero dati devono essere mantenuti in memoria, l'occupazione dovuta all'albero dati è lineare in accordo col fatto che la complessità dell'algorimo dipende dal numero di nodi della query.

Nella seconda fase, invece, viene valutata l'efficienza al crescere del numero di nodi della query conservando, per ognuna, lo stesso documento dati (440 nodi). Sono stati quindi generate 5 interrogazioni da rivolgere allo stesso documento XML. Tutte le interrogazioni sono costituite da elementi dell'esempio 4. L'interrogazione è pertanto volta alla ricerca dello stesso documento ma ognuna contiene un numero diverso di elementi quindi un diverso grado di precisione. La prima query considerata è composta da tre nodi, la seconda da 5 la terza da 7 (figura 5.3) la quarta da 9 e la quinta da 11 nodi.

I risultati di questa seconda fase di test, sono mostrati nelle figure 5.7 e 5.8.

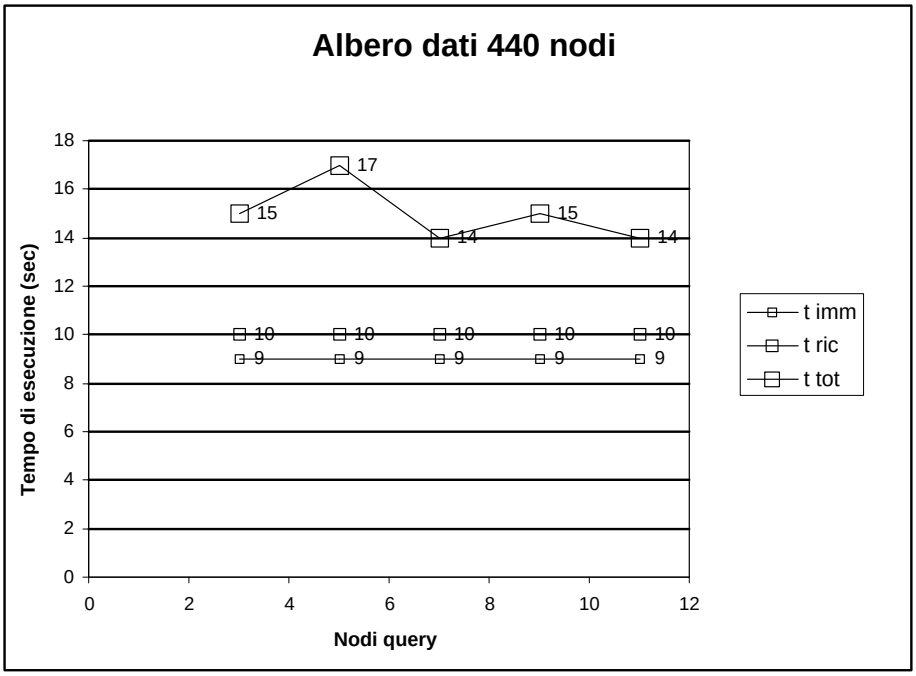


Figura 5.7: Test di scalabilità: tempi di esecuzione

Come messo in evidenza dai test:

- i tempi di risoluzione delle query presentano un andamento quasi costante all'aumentare del numero di nodi della query considerata. Questo è dovuto al fatto che il filtro riesce a contenere i gruppi di nodi che di volta in volta devono essere confrontati con l'albero query. Infatti, un numero di nodi query maggiore comporta una maggiore complessità (con andamento esponenziale) nell'algoritmo per il calcolo della tree edit distance ma,

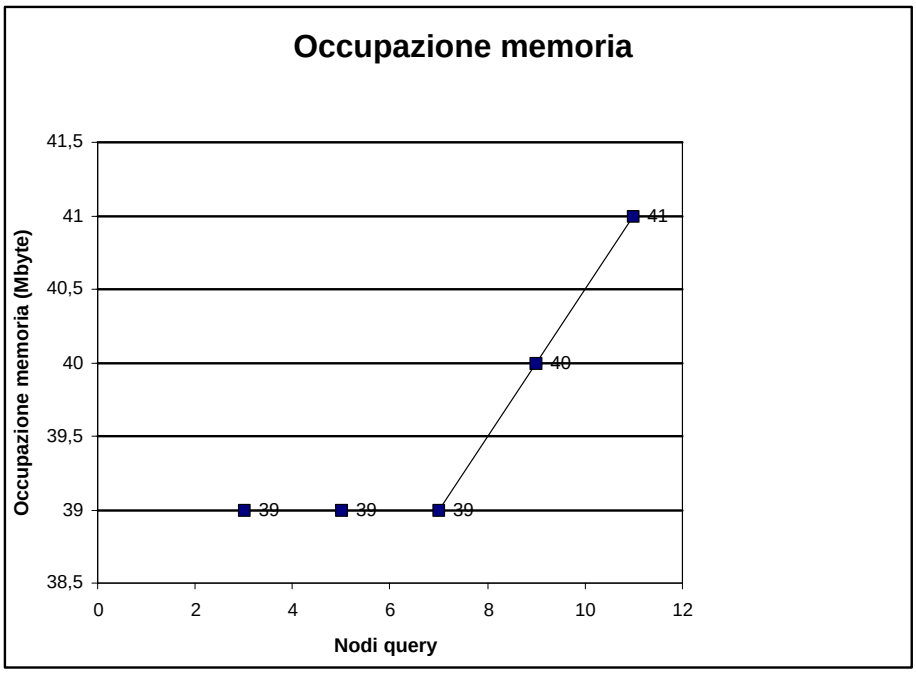


Figura 5.8: Test di scalabilità: occupazione memoria

comporta anche una maggiore selettività del filtro, che, traducendo il maggior numero di nodi query in criteri di selezione limita i gruppi di nodi da confrontare con l'albero query tramite l'algoritmo della tree edit distance;

- l'occupazione di memoria presenta un andamento di tipo esponenziale all'aumentare del numero di nodi della query in accordo col fatto che la complessità dell'algorimo dipende dal numero di nodi della query.

Precisiamo che tutti test effettuati sulla collezione *dblp* sono stati eseguiti sotto le seguenti condizioni:

- $soglia = 3$;
- $parentela = 2$;
- $ril\_par = 0$;

5.2.3 Variazioni dei parametri

Per mostrare gli effetti della variazione dei parametri abbiamo deciso di utilizzare la collezione *interna* in quanto è stata appositamente costruita per testare l'algoritmo ed esprime meglio le sue funzionalità.

I risultati mostrati fino ad ora per tale collezione, sono riferiti ai valori di default dei parametri *soglia* e *parentela*, rispettivamente fissati a 3 (distanza massima tra la query e le sue rapprsentazioni trovate nell'albero dati) e 2 (massimo rilassamento del vincolo di parentela padre-figlio).

Gli effetti della variazione di questi parametri sono stati misurati effettuando tre tipi di test i cui risultati sono mostrati nelle figure 5.9, 5.10 e 5.11, relativi, come sopra detto, alla collezione *interna*.

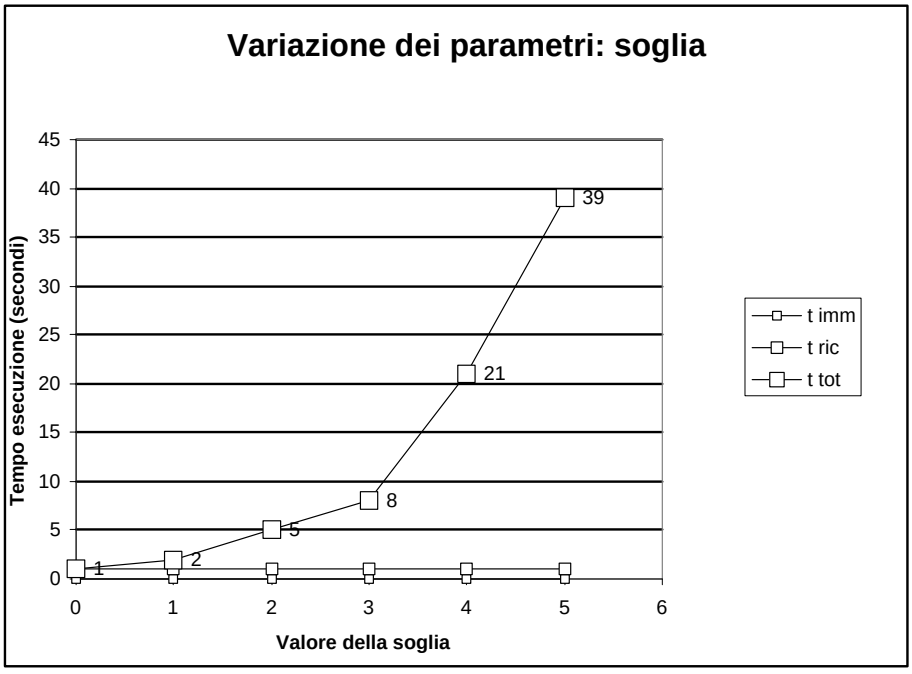


Figura 5.9: Efficienza al variare del valore di soglia

In questo primo grafico (figura 5.9) viene evidenziato come al crescere della soglia, aumentando le soluzioni accettabili, aumenta il tempo richiesto per la ricerca delle soluzioni stesse. Il tempo di esecuzione presenta un andamento più che lineare in quanto aumentanto la soglia il numero di gruppi di nodi che possono soddifare la query crescono in modo più che lineare. Ricordiamo infatti

che, le possibili combinazioni che si possono costruire su un insieme di elementi aumenta in modo esponenziale col numero di elementi stessi. In questo caso il numero di elementi è lo stesso ma aumentando la soglia permettiamo più libertà alle combianzioni che devono essere create e che quindi crescono comunque in modo più che lineare.

Il test di figura 5.9 è relativo sempre all'interrogazione Q in figura 5.2 eseguita sull'albero T di figura 5.1 sotto le seguenti condizioni:

- $parentela = 2$;
- $ril\_par = 1$;

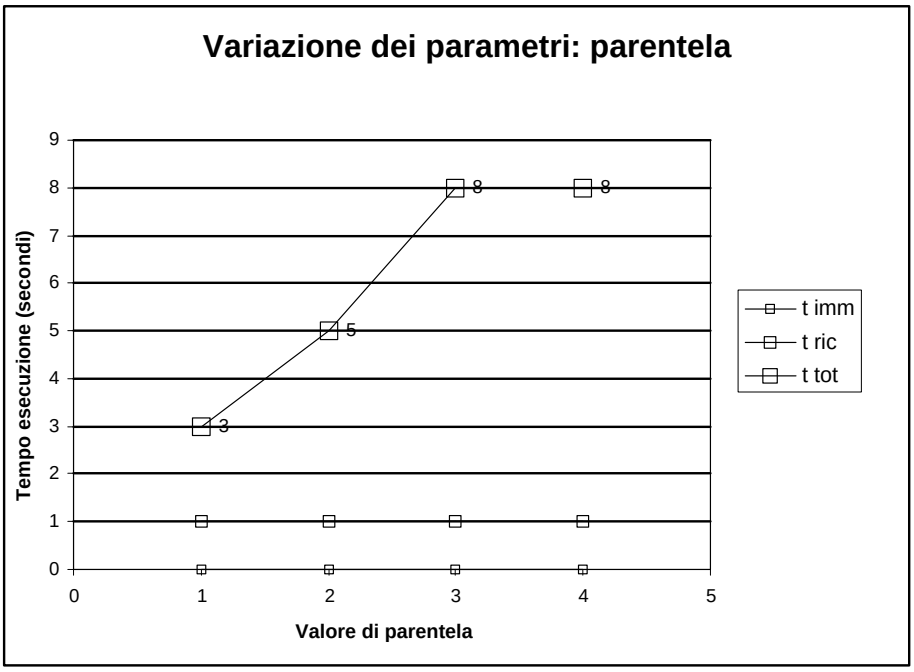


Figura 5.10: Efficienza al variare del valore di parentela

Anche nel grafico in figura 5.10 viene evidenziato come al crescere del valore *parentela*, aumentando le soluzioni accettabili, aumenta in modo più che lineare il tempo richiesto per la ricerca delle soluzioni stesse. Tuttavia in questo caso si osserva un livello di saturazione. Infatti mentre rilassando il valore della soglia il numero di soluzione cresce sempre più, rilassando il vincolo sulla parentela questo non si verifica. Per prima cosa, avendo associato un costo diverso da zero a ciascun rilassamento sui vincoli di parentela ($cost\_ril\_par \neq 0$) e fissato un valore di soglia, non è comunque possibile rilassare liberamente tali vincoli.

Inoltre la struttura dell'albero dati da noi considerato non presenta livelli di profondità tali da permettere molti rilassamenti.

Il test di figura 5.10 è relativo sempre all'interrogazione Q in figura 5.2 eseguita sull'albero T di figura 5.1 sotto le seguenti condizioni:

- $soglia = 3$;
- $ril\_par = 1$;

Per concludere vogliamo evidenziare l'efficacia del filtro realizzato mostrando l'andamento del numero di gruppi selezionati dal filtro ed il numero di quelli che effettivamente presentano una distanza inferiore alla soglia fissata al variara del valore di soglia (grafico in figura 5.11. Si può allora vedere come il filtro sia in grado limitare i gruppi di nodi dati da considerare limitando perciò la complessità del modello da noi progettato per il calcolo della tree edit distance.

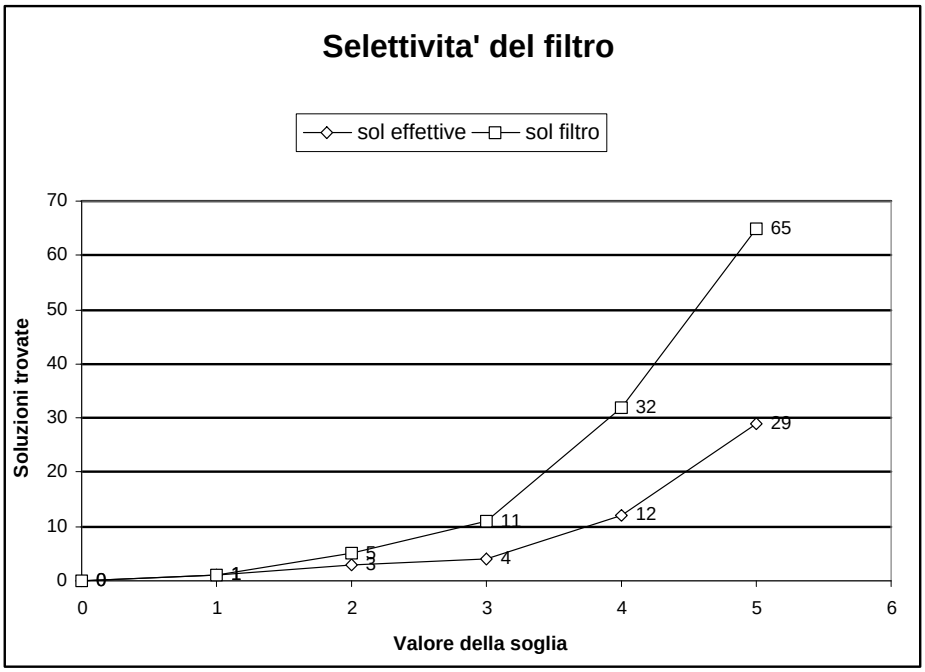


Figura 5.11: Selettività del filtro al variare della soglia

Il test di figura 5.11 è relativo sempre all'interrogazione Q in figura 5.2 eseguita sull'albero T di figura 5.1 sotto le seguenti condizioni:

- $parentela = 2$;
- $ril\_par = 1$;

Conclusioni e sviluppi futuri

In questo lavoro sono stati raggiunti i seguenti risultati:

- è stata definita una *metrica di similarità* tra alberi, basata sul concetto di *Tree Edit Distance*, in grado di esprime adeguatamente ed *efficacemente* il grado di somiglianza tra due alberi.
- utilizzando questa metrica, è stato definito e affrontato il problema di *risoluzione di una query* inteso come, dato un albero query ed un albero dati di modeste dimensioni, individuare le soluzioni la cui distanza dalla query non supera una data *soglia*;
- è stato definito un filtro in grado di ricercare le parti di un albero dati di grandi dimensioni simili all'albero query basando la ricerca sul contenuto espresso dalle label dei nodi appartenenti ai due alberi. Solo in parte si è modificato il filtro, inserendo criteri di selettività basati sulla struttura, al fine di migliorare e ridurre il numero delle soluzioni trovate.
- è stato infine progettato un ambiente comune che riunisce tutte queste funzionalità e ne permette un utilizzo semplice e congiunto, permettendo all'algoritmo di risolvere query anche su grandi quantità di dati e garantendo all'utente:
  - il riconoscimento di tutte le soluzioni in tempi modesti.
  - la possibilitaà di configurare i vari parametri.

In questo modo si è ottenuto un ambiente completo, progettato e realizzato in modo da essere di interesse dal punto di vista della ricerca scientifica, proponendo soluzioni riguardanti tematiche poco affrontate.

Per quanto riguarda gli sviluppi futuri, sarebbe interessante:

- ricercare criteri per la costruzione dei marking che limitino le combinazioni possibili. Alcune proposte sono già state suggerite da Shasha in [13] ma nessuna di queste garantisce di trovare il miglior marking legale.

- potenziare gli algoritmi proposti per la realizzazione del filtro, in particolare aggiungendo ulteriori criteri di selezione in modo da perfezionarne l'efficienza. Infatti per migliorare le prestazioni dei tempi di ricerca sarebbe necesario migliorare ulteriormente l'efficacia della ricerca delle sottoparti, lavorando ancora sui filtri proposti e studiandone di nuovi al fine di riperire tutte e solo le sottoparti che risultano poi effettivamente avere una distanza dalla query inferiore alla soglia.

- studiare nuovi algoritmi per il calcolo della tree edit distance tra alberi unordered in grado di limitare la complessità computazionale che caratterizza il modello dell'algoritmo da noi seguito. Come si capisce, infatti, il problema del calcolo della distanza tra alberi è un problema di grande interesse non solo nell'ambito della interrogazioni su dati strutturati, ma per una grande vastità applicazioni che vanno dalla biologia molecolare alla visione artificiale.

- migliorare le rappresentazioni degli alberi introducendo una distinzione, che in questo lavoro è assente, tra la rappresentazione di alberi dati ed alberi query. Occorre infatti osservare che mentre per i nodi dell'albero dati sono sufficienti poche informazioni, per l'albero query sono invece necessarie molte più informazioni.

- migliorare le prestazioni della funzione che converte la forma a stringa nella struttura ad albero utilizzata dall'algoritmo. Come si vede dal grafico in figura 5.5 il più del tempo di risoluzione della query è impiegato per convertire il documento nella struttura dati che adotta l'algoritmo.

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, confrontandosi direttamente con esigenze e problematiche non solo di correttezza, ma anche

di efficienza. Si è avuta la possibilità di progettare un software di notevoli dimensioni e di curarne lo sviluppo, dalle funzionalità principali alla gestione dei formati dei documenti in ingresso, approfondendo tra l'altro il linguaggio di programmazione Java e gli strumenti messi a disposizione dal nuovo standard XML quali XPath. Lo stesso campo di applicazione, quello della ricerca di similarità tra alberi, è stato un argomento di studio affascinante: la ricerca in questo contesto è specialmente attiva ed entrare a farne parte proponendo risultati nuovi e, ci auguriamo, di interesse è stato sicuramente un motivo di grande soddisfazione.

Bibliografia

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *In proceedings of the International conference of extending database technology*, pages 496–513, 2002.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[3] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1), 2000.

[4] P. Ciaccia and W. Penzo. Relevance ranking tuning for similarity queries on XML data. In *Proc. of the 28th VLDB conference, Hong Kong, China*, 2002.

[5] World Wide Web Consortium(W3C). Extensible Markup Language (XML). http://www.w3.org/XML/.

[6] M. Fernandez, J. Siméon, and P. Wadler. XML and query languages: experiences and examples. In FST TCS, Delhy, Dicember 1999.

[7] P. Kilpeläinen. Tree matching problems with applications to structured database. Report A-1992-6, University of Helsinky, Finnland, November 1992.

[8] P. N. Klein. Computing the edit distance between unrooted ordered trees, 1998.

[9] K.Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information processing letters*, 42, 1992.

[10] V. Levenshtein. Binary codes capable of correcting insertions, deletions and reversals. In *Cybernetics and control theory*, pages 707–710, 1966.

[11] T. Schlieder. Schema-driven evaluation of approximate tree-pattern queries, 2002.

[12] T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In proceedings of the ACM SIGIR workshop on XML and information retrieval, Athene, 2000.

[13] D. Shasha, J.T. Wang, K. Zhang, and F.y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE transactions on system, man, and cybernetics*, 24(4):668–678, 1994.

[14] D. Shasha, J.T.L. Wang, and R. Giugno. Algorithmics and applicatios of tre and graph searching, 2002.

[15] Vari. DBLP Database. http://www.informatik.uni-trier.de/ley/db.

[16] K. Zhang and D. Shasha. Tree pattern matching. Pattern matching algorithms, Apostolico and Galil Editors, Oxford University Press, 1997.