XML Covering Index utilizzando la relazione di Simulazione

Luca Casini
Università degli studi di Perugia
Corso di Laurea in Informatica

10 aprile 2015

Indice

Capitolo 1

Nozioni Introduttive

Iniziamo introducendo alcune nozioni alla base di questa tesi: i **grafi**, strutture matematiche utilizzate nella rappresentazione dei documenti XML, e le relazioni di **Simulazione**, **Bisimulazione** e **Trace Equivalence**, relazioni binarie sui grafi che possono essere usate nella creazione di indici XML.

1.1 I Grafi

Introduciamo formalmente il concetto di grafo e le notazioni che utilizzeremo nel corso della tesi.

I grafi sono strutture matematiche di grande importanza in molteplici campi scientifici. Un grafo è composto da un insieme di entità astratte chiamate **nodi** e un insieme di coppie di nodi, detti **archi**, che sono in relazione tra loro.
Possiamo rappresentare graficamente un grafo come dei punti nello spazio, i nodi, connessi da delle linee, gli archi.

Definizione 1 (Grafo)**.** *E' chiamato Grafo una coppia di insiemi* $G = (N, E)$ *dove* N *è un insieme non vuoto di elementi detti nodi e* $E \in N \times N$ *è l'insieme degli elementi detti archi. Chiameremo* $|N|$ ***ordine*** *del grafo e* $|E|$ ***dimensione*** *del grafo.*

Se esiste in E un insieme di archi che ci permette di raggiungere un nodo v partendo da un altro nodo u diremo che esiste un cammino tra i due nodi.

Definizione 2 (Cammino)**.** *dati due nodi* $u, v \in N$ *esisterà un cammino u v di lunghezza k se esiste una sequenza* $< v_0, v_1, v_2, ..., v_k >$ *di vertici tale che la coppia* $(v_(i-1), v_i) \in E$ *per* $i = 1, .., k$*, con* $u = v_0$ *e* $v = v_k$*.*

In base alle proprietà che verificano possiamo dividere i grafi nel seguente modo:

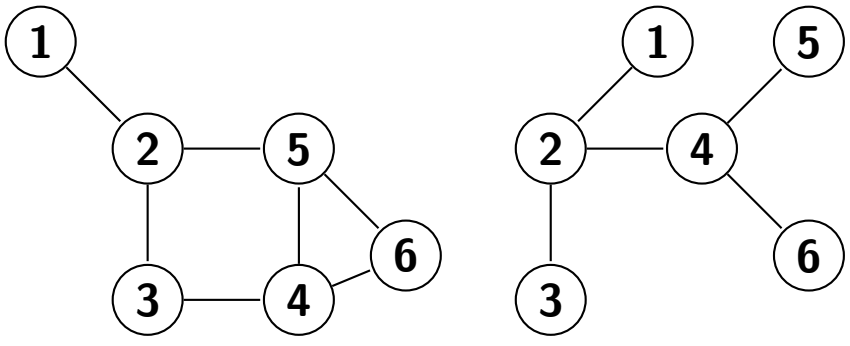
- un grafo può essere **finito o infinto** a seconda della cardinalità di N.
- un grafo si dice **etichettato** se ad ogni nodo e/o vertice è associata una etichetta di qualche tipo.
- un grafo è **aciclico** se e soltanto se non esiste un cammino che riporti al nodo di partenza, ossia $\nexists u \in N :< u, u >\in E$.
- un grafo è detto **orientato** nel caso in cui gli archi abbiano un verso di percorrenza; un grafo semplice è non orientato ovvero $(u, v) \in E \implies (v, u) \in E$.

Definizione 3 (Grafo fortemente connesso)**.** *Un grafo orientato è* ***fortemente connesso*** *se due vertici qualsiasi sono raggiungibili l'uno dall'altro, ovvero per ogni* $u, v \in N$*, vale* $u \rightsquigarrow v$*.*

Definizione 4 (Componente fortemente connessa)**.** *Una* ***componente fortemente connessa*** *di un grafo orientato* G *è un sottografo massimale di* G *fortemente connesso in cui esiste un cammino orientato tra ogni coppia di nodi ad esso appartenenti.*

Le componenti fortemente connesse formano una partizione di G poiché un nodo non può trovarsi contemporaneamente in due componenti fortemente connesse, di conseguenza un grafo orientato è fortemente connesso se e solo se ha una sola componente connessa In seguito illustriamo alcuni esempi con i vari tipi di grafi:

Esempio 1

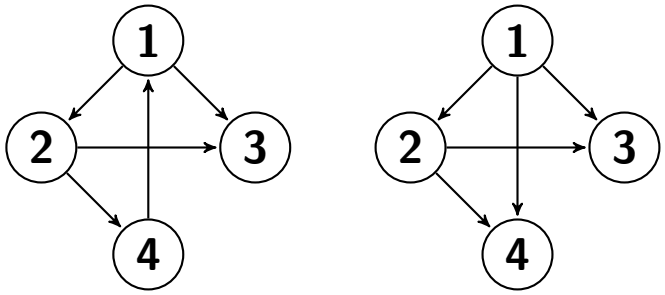


(a) Grafo ciclico non orientato

(b) Grafo aciclico non orientato

Figura 1.1: *Grafi ciclici/aciclici non orientati etichettati sui nodi*

Esempio 2



(a) Grafo ciclico orientato che contiene il ciclo {1,2,4,1}

(b) Grafo aciclico orientato

Figura 1.2: *Grafi ciclici/aciclici orientati etichetatti sui nodi*

Esempio 3

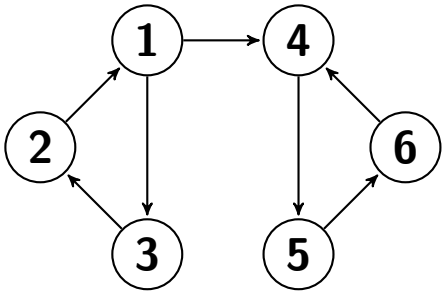


Figura 1.3: *Un grafo orientato con due componenti fortemente connesse {1,2,3} e {4,5,6}*

1.1.1 Alberi

Questa tesi si occuperà di un tipo particolare di grafo detto **albero**. Un albero è un grafo aciclico che verifica certe proprietà; questo nome deriva dal fatto che, come un vero albero, possiede una radice da cui partono delle ramificazioni (sotto-alberi).

Definizione 5 (Albero)**.** *Si dice albero un grafo G che verifica una della seguenti condizioni equivalenti:*

- *G è aciclico e connesso.*
- *aggiungere un arco a G crea un ciclo.*
- *se si rimuovere un arco G non è più connesso.*

La struttura dati comunemente utilizzata in informatica è una tipologia di albero detto radicato *radicato*, cioè un albero in cui un nodo viene scelto come radice.

Definizione 6. *Definiamo di seguito la radice e tutti i componenti dell'albero a partire da essa:*

Radice: *La radice è il nodo di partenza. Tra la radice e qualsiasi nodo dell'albero c'è un percorso univoco (il senso del percorso è dato dall'orientamento dell'albero, generalmente è dalla radice al nodo).*

Livello: *Il livello è dato dalla lunghezza del percorso dal nodo alla radice, la radice ha livello 0. Il livello massimo raggiunto è detto* ***altezza*** *dell'albero. I nodi allo stesso livello sono detti* ***cugini****.*

Discendente/Antenato: *Dato un nodo i suoi discendenti sono tutti i nodi da esso raggiungibili allontanandosi dalla radice. Muovendoci verso la radice troveremo i suoi Antenati.*

Figlio/Padre: *Dato un nodo i suoi figli sono i discendenti al livello inferiore. Il padre di un nodo è l'antenato al livello superiore. I nodi figli dello stesso padre si dicono* ***fratelli****.*

Foglia: *Un nodo è una foglia se non ha figli. E,->,>=' anche detto* ***nodo esterno****.*

Esempio 4

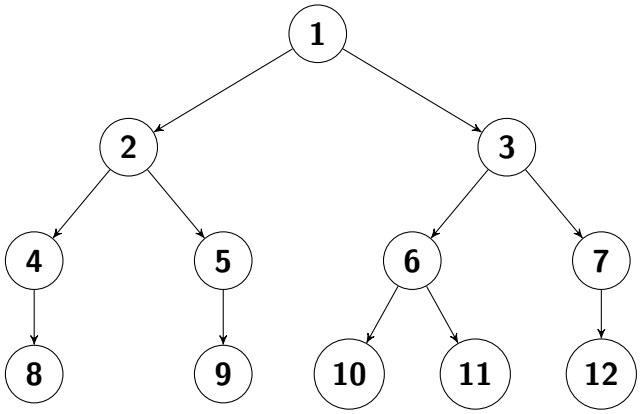


Figura 1.4: *Esempio di Albero*

Nella figura 1.4 vediamo un esempio di albero. Il nodo 1 è la radice, i nodi 8,9,10,11,12 sono foglie al livello 3. Il nodo 3 è padre del nodo 6 e 7 e antenato di 6,7,10,11,12.

1.2 Simulazione, Bisimulazione e Trace Equivalence

In questa sezione introduciamo la relazione d'equivalenza detta simulazione e due relazione ad essa collegate: la bisimulazione e la trace equivalence.

Le classi di equivalenza che costruiamo grazie a queste relazioni possono essere utilizzate per creare un grafo ridotto detto **grafo quoziente.** L'uso del grafo quoziente permette, a fronte di una perdita di informazione, di lavorare con strutture più piccole risparmiando tempo e risorse.

1.2.1 Simulazione

Iniziamo definendo formalmente la relazione di simulazione, in seguito illustreremo un esempio su un albero.

Definizione 7. *Sia* $G = (V, E, A, \langle\langle . \rangle\rangle)$ *un grafo etichettato dove:*

- V *è l'insieme dei vertici.*
- $E \subseteq V^2$ *è l'insieme dei cammini.*

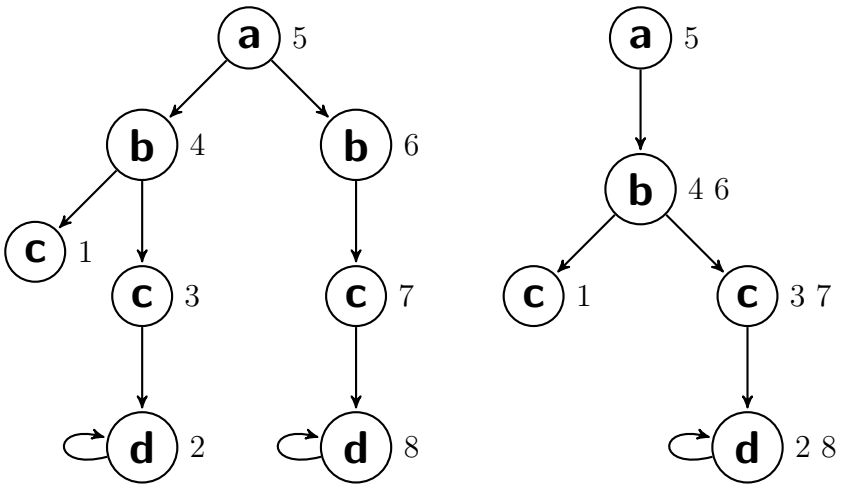
- *A è l'insieme delle etichette.*
- $\langle\langle . \rangle\rangle : V \to A$ *è una funzione che assegna ad ogni vertice* v *la propria etichetta* $\langle\langle v \rangle\rangle$.

Data la funzione $post(v) = \{u | (v, u) \in E\}$ *che restituisce l'insieme dei successori del vertice* v. *Possiamo definire una relazione binaria* $\preceq \subseteq V^2$ *detta* simulazione *se si verificano le seguenti condizioni:*

1. $\langle\langle u \rangle\rangle = \langle\langle v \rangle\rangle$
2. $\forall u' \in post(u) \exists v' \in post(v) | u' \preceq v'$

Lemma 1. *Due vertici* u *e* v *si dicono simili* $u \approx^S v$ *se* $u \preceq v$ *e* $v \preceq u$; *la* similarità $\approx^S$ *è una relazione di equivalenza.*

Esempio 5



(a) Grafo G **(b)** Grafo $G/_{\simeq^s}$

Figura 1.5: *Esempio di Simulazione.*

A sinistra abbiamo il grafo di partenza e a destra il grafo quoziente ricavato dalla relazione $\simeq^S$. Da notare come la cardinalità sia quasi dimezzata nel grafo quoziente.

1.2.2 Bisimulazione

Definiamo ora la relazione di bisimulazione

Definizione 8. *Dati due vertici* u *e* v, $u \simeq v$ *se:*

- $\langle\langle u \rangle\rangle = \langle\langle v \rangle\rangle$
- $\forall u' \in post(u) \exists v' \in post(v) | u' \simeq v'$
- $\forall v' \in post(v) \exists u' \in post(u) | v' \simeq u'$

Diremo quindi che u bisimula v; se $u \simeq v$ *e* $v \simeq u$ *allora si ha una relazione di* bisimulazione $\approx^B$.

Il grafo quoziente $G/_{\approx^B}$ conserva un linguaggio più espressivo rispetto a quello ottenuto sfruttando la simulazione ma, essendo una relazione più fine, non fornisce una riduzione altrettanto importante in termini di spazio. Va notato come grazie all'algoritmo di Paige-Tarjan è possibile calcolare la bisimulazione in $O(mlogn)$.

1.2.3 Trace Equivalence

Illustriamo infine la trace equivalence, in figura possiamo vedere un confronto tra le due relazioni.

Definizione 9. *Dati due vertici u e v, diciamo che u domina v se:*

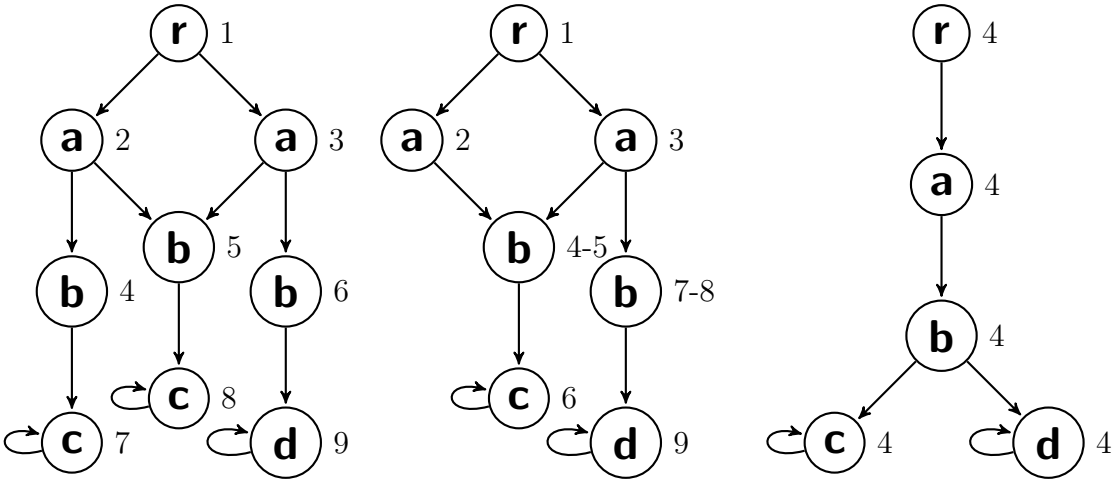
- *definito* $\bar{u}$ *un cammino con radice in u*
- $\forall\, \bar{u} \quad \exists\, \bar{v} \mid \langle\langle \bar{u} \rangle\rangle = \langle\langle \bar{v} \rangle\rangle$

TraceG La relazione di equivalenza $\approx^T$ *è detta Trace Equivalence e vale se u domina v e v domina u.*

Lemma 2. *La trace equivalence è implicata dalla simulazione ma non viceversa.*

Questa relazione è molto importante perché due vertici equivalenti soddisfano le stesse formule di LTL e quindi il grafo quoziente $G/_{\approx^T}$ può essere usato per verificarle; tuttavia costituire questo grafo è molto difficile, il problema infatti è PSPACE-completo.

Esempio 6



(a) Grafo G **(b)** Grafo $G/_{\approx^B}$ **(c)** $G/_{\approx^T}$

Figura 1.6: *Esempi di Trace Equivalence e Bisimulazione.*

A sinistra abbiamo un grafo G di partenza, al centro il grafo quoziente $G/_{\approx^B}$ e a destra il grafo quoziente $G/_{\approx^T}$. Notare come la Bisimulazione non ha grandi capacità di riduzione rispetto a trace equivalence e simulazione.

Capitolo 2

Calcolo della Simulazione

Negli anni sono stati proposti vari algoritmi per il calcolo della simulazione, più o meno efficienti. Questa tesi si basa sul lavoro di **M.Henzinger, T.Henzinger e P.Kopke** che nel 1995 hanno proposto degli algoritmi efficienti per il calcolo della simulazione su grafi finiti e infiniti.
L'algoritmo HHK, partendo da un implementazione naif del problema (che termina in $O(m^2n^3)$), viene rifinito in due passi fino a terminare in tempo $O(mn)$. Di seguito ne descriviamo il funzionamento e ne analizziamo complessità e correttezza. In figura viene mostrato lo pseudo-codice della versione più efficiente.

2.1 L'Algoritmo HHK

Nella sua versione Naif l'algoritmo costruisce un insieme $sim(v)$ per ogni nodo v e poi confronta i successori di v con i successori dei suoi simulatori effettuando una raffinamento di $sim(v)$. Quando non è più possibile raffinare il nessun insieme l'algoritmo termina.

L'algoritmo efficiente inizia con un ciclo `for` in cui si costituisce per ogni vertice $v \in V$: l'insieme $prevsim(v)$ che conterrà i simulatori canditati di v nell'iterazione precedente e viene inizializzato uguale a V, $sim(v)$ che conterrà gli effettivi simulatori di v e viene inizializzato con tutti i vertici che hanno la stessa etichetta di v, l'insieme $remove(v)$, contente i predecessori dei simulatori che sono stati eliminati, che verrà usato per raffinare l'insieme dei simulatori di ogni vertice.
All'interno del `if` si fa uso della funzione $post(v)$ definita come $\{u|(v,u) \in E\}$, ossia l'insieme di tutti i successori di v.
Nel ciclo `while` invece vengono presi in considerazione, in maniera non de-

terministica, tutti i vertici v per cui $remove(v) \neq \emptyset$. Si controlla che i predecessori u di v non siano simulati da vertici $w \in remove(v)$, in caso contrario vengono rimossi dai simulatori di u. A fine ciclo viene aggiornato $prevsim(v)$ e viene svuotato $remove(v)$, il while termina quando $remove(v) = \emptyset$ per tutti i vertici del grafo.

2.1.1 Pseudocodice

Mostriamo di seguito lo pseudocodice della versione naif e della versione più raffinata dell'algoritmo HHK.

Algorithm 1 Naif HHK

1: **for all** $v \in V$ **do** $sim(v) := \{u \in V | l(u) = l(v)\}$
2: **while** there are three vertices u, v, w such that
3: $v \in post(u)$,
4: $w \in sim(u)$,
5: $post(w) \cap sim(v) = \emptyset$ **do**
6: $sim(u) \leftarrow sim(u) \ \{w\}$

Algorithm 2 Efficent HHK

1: **for all** $v \in V$ **do**
2: $prevsim(v) \leftarrow V$
3: **if** $post(v) = \emptyset$ **then** $sim(v) := \{u \in V | l(u) = l(v)\}$
4: **else** $sim(v) := \{u \in V | l(u) = l(v)$ **and** $post(u) \neq \emptyset\}$
5: $remove(v) := pre(V) \setminus pre(sim(v))$
6: **while** there is a vertex $v \in V$ such that $remove(v) \neq \emptyset$ **do**
7: **Assert:** $\forall\ v \in V\ remove(v) = pre(prevsim(v)) \setminus pre(sim(v))$
8: **for all** $u \in pre(v)$ **do**
9: **for all** $w \in remove(v)$ **do**
10: **if** $w \in sim(v)$ **then** $sim(u) \leftarrow sim(u) \setminus w$;
11: **for all** $w' \in pre(w)$ **do**
12: **if** $post(w') \cap sim(u) = \emptyset$ **then** $remove(u) \leftarrow remove(u) \cup w'$
13: $prevsim(v) := sim(v)$
14: $remove(v) := \emptyset$

2.1.2 Correttezza e Complessità

La versione Naif ha un costo computazionale pari a $O(m^2n^3)$, il costo è dominato dal ciclo `while`.

L'algoritmo Efficent HHK termina in tempo $O(mn)$, la complessità si compone nel modo seguente: l'inizializzazione di $sim(v)$ richiede tempo $O(n^2)$(ricordiamo che $n \leq m$); l'inizializzazione di $remove(v)$ per ogni v richiede tempo $O(mn)$. Dati due vertici v e w, se la condizione $w \in remove(v)$ è vera all' iterazione i del ciclo `while` sarà allora falsa per ogni iterazione $j \geq i$.
Abbiamo infatti che: (1)in tutte le iterazioni $w \in remove(v)$ implica che $w \notin pre(sim(v))$. (2)il valore di $prevsim(v)$ nell'iterazione j è un sottoinsieme di $sim(v)$ nell'iterazione i. (3)la condizione in `assert` all'inizio del ciclo `while`. Segue che il ciclo `forall` $w \in remove(v)$ viene eseguito $\Sigma_v\Sigma_w(|pre(v)| = O(mn))$ volte.
La condizione $w \in sim(u)$ è verificata al più una volta per ogni w e u, poiché se è vera per qualche w questo viene rimosso definitivamente da $sim(u)$. Per cui l'`if` più esterno all'interno del ciclo `while` contribuisce con un tempo $\Sigma_w\Sigma_u(1 + |pre(v)| = O(mn))$. Così otteniamo un tempo totale di $O(mn)$.

2.1.3 Algoritmo in Azione

Mostriamo come opera l'algoritmo nella sua versione naif con un esempio su un semplice grafo.

Esempio 7

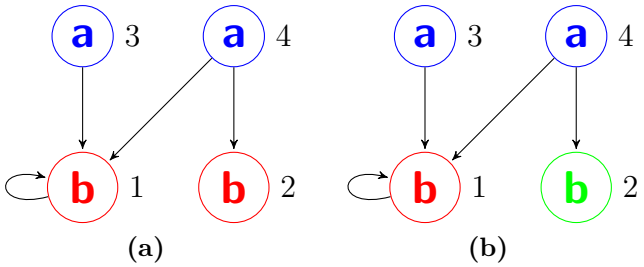


Figura 2.1: *L'algoritmo HHK in azione.*

All'inizio l'algoritmo crea per ogni vertice un insieme contenente tutti i verti con la stessa etichetta avremo dunque:

1. $sim(1) = 1, 2$

2. $sim(2) = 2, 1$

3. $sim(3) = 3, 4$

4. $sim(4) = 4, 3$

entrando nel ciclo while prendiamo in considerazione i suoi successori di nodo e i successori dei suoi simulatori. Se l'intersezione tra questi due insiemi e vuota per qualche simulatore del nodo allora questa viene eliminato dall'insieme dei simulatori.
Nel nostro caso abbiamo:

1. $post(1) = 1$

2. $post(2) = \emptyset$

3. $post(3) = 1$

4. $post(4) = 1, 2$

Partiamo con il nodo 3 e verifichiamo il suo simulatore 4, dall'algoritmo risulta che $post(4) \cap sim(1) = \{1, 2\} \cap \{1, 2\} =\not\emptyset$ quindi la simulazione è mantenuta.
prendiamo il nodo 1 e il suo nodo simulatore 2, abbiamo che $post(2) \cap sim(1) = \emptyset \cap \{1, 2\} = \emptyset$ per cui il nodo 2 viene eliminato dai simulatori di 1.

Capitolo 3

XML - eXtensible Markup Language

In questo capitolo verrà descritto il linguaggio XML. Partendo dalla sua storia, dall'idea di partenza alle successive evoluzioni che hanno portato allo standard attuale, si illustreranno le caratteristiche che lo contraddistinguono, la sintassi e la semantica. Verranno introdotti anche gli strumenti necessari per definire e manipolare un documento XML come DTD, XSD e XPath. In conclusione una piccola parentesi su XML e il modello relazionale.

3.1 Breve Storia di XML

La storia di **XML** *(eXtensible Markup Languages)* inizia negli anni '70 con lo sviluppo di **SGML** *(Standardised Generalised Markup Language)* da parte di Charles Goldfarb, insieme a Ed Mosher e Ray Lorie mentre lavoravano per IBM (Anderson, 2004). SGML a dispetto del nome non è propriamente un linguaggio di markup, ma piuttosto un linguaggio usato per specificare linguaggi di markup. Lo scopo di SGML era quello di creare vocabolari da poter usare nel markup di documenti con tag strutturati. Si immaginava all'epoca che certi documenti leggibili dalle macchine dovessero restare tali per decenni. Una delle applicazioni più popolare di SGML arrivò con lo sviluppo di **HTML** *(HyperText Markup Language)* da parte di Tim Berners Lee alla fine degli anni '80 (Raggett, Lam, Alexander, Kmiec, 1998). Fin dal suo sviluppo HTML divenne in qualche modo vittima della propria popolarità,venendo rapidamente adottato ed esteso in molti modi oltre la visione originale.
Resta popolare tutt'oggi come tecnologia di presentazione ma è considerato inadatto come formato generico per immagazzinare dati. Quando si tratta di scambiare e immagazzinare dati HTML è una cattiva scelta, essendo origina-

riamente indirizzato alla presentazione, mentre SGML è considerato troppo complicato per l'uso comune.
XML riempe questa lacuna risultando leggibile sia alle macchine che alle persone, restando flessibile abbastanza da supportare scambi di dati indipendenti dalla piattaforma e dall'architettura. Alla base, XML permette ad un ingegnere del software di creare un vocabolario e di utilizzarlo per descrivere dei dati. Ad esempio, nello scambio di dati tra computer, il numero 42 non è significativo a meno che non si trasmetta pure il significato dei dati, come la temperatura del processore espressa in gradi Celsius.
Solo quando mittente e destinatario sono in accordo sul significato dei dati possono farne qualcosa di utile. Prima dello sviluppo di XML, era in qualche modo necessario un accordo tra sistemi a priori sui dati e il loro significato. Con lo sviluppo di XML i dati possono essere scambiati senza accordi precedenti fintanto che entrambi i sistemi dispongano dello stesso vocabolario, ossia, parlino lo stesso linguaggio.
Dallo sviluppo di XML sono emerse diverse applicazioni nelle seguenti aree:

Pubblicazioni Web: XML permette di creare pagine interattive, dando possibilità di personalizzazione al cliente e rende la creazione di applicazioni di e-Commerce più intuitivo.

Ricerche Web e automazione: XML definisce il tipo di informazione contenuto in un documento, rendendo più facile la restituzione di risultati utili durante le ricerche Web.

Applicazioni Generiche: XML fornisce un metodo standard per l'accesso alle informazioni, rendendo più facile ad applicazioni e dispositivi di ogni tipo utilizzare, conservare, trasmettere e presentare dati.

Applicazioni e-Business: Implementazioni XML rendono l'*Electronic Data Interchange* (EDI) più accessibile per lo scambio di informazioni, le transazioni tra azienda e azienda e le transazioni tra azienda e consumatore.

Metadata Applications: XML rende più semplice esprimere metadati in un formato portabile e riusabile.

Computing Pervasivo: XML fornisce tipi di dati portabili e strutturati da presentare su dispositivi pervasivi (wireless) come notebook, tablet e smartphone.

3.2 Struttura e Sintassi

Tutti i documenti XML sono formati da elementi rappresentati da due tag, uno di apertura e uno di chiusura, gli elementi possono contenere dati, altri elementi, una combinazione dei due, o nessuno di essi(elementi vuoti). I Documenti iniziano con un unico elemento radice che contiene sotto-elementi, i loro sotto-elementi e cosi via. Questo risulta in una struttura gerarchica ad albero all'interno del documento.

Esempio 8 Ecco un esempio di codice XML

```
<root>
  <element attribute1="value">content</element>
  <element attribute1="value">content</element>
  <element2 attribute2="3">
    <sub-element>content</sub-element>
    <sub-element>content</sub-element>
   </element>
</root>
```

3.2.1 Definire la struttura dei dati: DTD e XSD

Pur essendo possibile intuire la struttura dei dati rappresentati leggendo il documento XML a volte è necessario definire una struttura rigida che vada rispettata; ci sono vari modi di ottenere questo, i più comuni sono *Data Type Definition* (DTD) e *XML Schema Definition* (XSD).

DTD - Data Type Definition: DTD fu introdotto durante lo sviluppo di SGML ed è per questo motivo più indicato per applicazioni indirizzate ai documenti, come HTML. HTML infatti è specificato usando DTD. Basato sulla *Extended Backus-Naur Form* (EBNF) può definire la struttura di un documento ma non ha l'abilità di specificare regole da applicare ai dati. Ovvero tutti i dati contenuti all'interno del documento vengono trattati dal DTD come una stringa. Questo si adatta ai linguaggi di markup per documenti, ma non è conveniente quando un'applicazione necessita il controllo sui dati contenuti.

Esempio 9 Un esempio di DTD

```
<!ELEMENT element (#PCDATA)*>
<!ELEMENT element2 (sub-element+)>
<!ELEMENT sub-element (#PCDATA)>
<!ATTLIST element
        attribute1 CDATA #REQUIRED>
<!ATTLIST element2
        attribute2 CDATA #REQUIRED>
```

XSD - XML Schema Definition: XSD fu sviluppato per ovviare a questa lacuna (W3C, 2000). Basato sulla stessa sintassi di XML, XSD definisce molti più tipi di dato e permette di specificare regole per verificare non solo alla struttura del documento XML, ma anche ai dati contenuti. In questo modo è possibile definire un tag XML con un tipo *nonNegativeInteger*,e un parser XML validante restituirebbe un errore se il tag presentasse dati diversi da un intero maggiore di zero.

Esempio 10 Esempio di XML Schema.

```
<xs:schema attributeFormDefault="unqualified"
        elementFormDefault="qualified"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="element"
                        maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string"
                                    name="attribute1" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="element2">
          <xs:complexType>
```

```
20              <xs:sequence>
21                <xs:element type="xs:string" name="sub-element"
22                        maxOccurs="unbounded" minOccurs="0"/>
23              </xs:sequence>
24              <xs:attribute type="xs:byte" name="attribute2"/>
25            </xs:complexType>
26          </xs:element>
27        </xs:sequence>
28      </xs:complexType>
29    </xs:element>
30  </xs:schema>
```

3.2.2 Rappresentazione e Manipolazione di file XML

All'interno di applicazioni i dati XML vengono rappresentati secondo il Document Object Model (DOM) e manipolati tramite API specifiche. La struttura del DOM è schematizzabile come un grafo: I tag (e gli attributi) corrispondo ai nodi, le relazioni gerarchiche corrispondo agli archi; l'assenza di attributi idref determina una rappresentazione ad Albero. Sebbene molto potente questo tipo di rappresentazione tuttavia non è agevole da gestire ed inoltre i dati XML non sono sempre disponibili direttamente. A volte infatti i dati sono troppo grandi per la memoria centrale (es.Database) oppure devono essere acceduti da remoto o possono provenire da un'altra applicazione.
E' emersa allora la necessità di sviluppare strumenti l'esplorazione, l'interrogazione e la manipolazione dei file XML. I più comuni e diffusi sono *XPath* e *XSLT*.

XPath è un linguaggio pensato navigare un documento XML ed ottenere le informazioni necessarie specificate nella query; nella sezione successiva lo introdurremo in dettaglio.

XSLT, eXtensible Stylesheet Language Transformation , è un linguaggio che si appoggia ad XPath e consente di manipolare le informazioni contenute in documento XML e di ottenere un altro documento che può essere di tipo XML o, molto comunemente, di tipo HTML. Viene spesso utilizzato per generare pagine dinamicamente partendo da una base di dati immagazzinata in un file XML.

3.3 XPath

XPath è un linguaggio che permette di navigare un documento XML in maniera simile alle directory del file system di Unix. É stato sviluppato per essere semplice così da essere integrato all'interno di linguaggi più specifici come XQuery, XSLT, XSD, XPointer, ecc.

La sintassi, che può essere estesa o abbreviata, si basa su delle espressioni dette Path Expressions, espressioni che definiscono un percorso verso un nodo o un insieme di nodi; questo percorso può essere assoluto (dalla radice a un nodo) o relativo (dal nodo corrente a un nodo).

Le Path Expressions sono divise in elementi separati dal carattere / detti **location steps**, ogni location step è composto da tre parti fondamentali: un **asse** (Axis Specifier), un **test sul nodo** (Node Test), un **predicato** (Predicate).
Un asse è un'indicazione sulla direzione da percorrere lungo l'albero che rappresenta il documento xml. Gli assi disponibili sono quelli in Tabella 3.1.

Un Node Test può essere semplicemente il nome del nodo oppure uno dei seguenti: comment() ottiene i nodi che contengono commenti, text() ottiene i nodi che contengono testo ossia le foglie dell'albero, processing-instrucition() restituisce i nodi che contengono codice come <?php echo $x ?>, node() restituisce ogni nodo.

I predicati o filtri sono delle espressioni scritte all interno di parentesi quadre che servono a selezionare soltanto i nodi che verificano la condizione che indicata. Per esempio `a[@href=www.google.it]` seleziona tutti i nodi `a` dove l'attributo `href` è uguale a `www.google.it`.

Le query XPath possono essere divise in due macro-categorie (Mohammad, Martin), Path Queries e Twig Queries. Una Path Query è una query che può essere risolta seguendo dei semplici percorsi nell'albero. Una twig query invece implica la valutazione di percorsi ramificati nel grafo XML per ottenere la risposta desiderata e pur aggiungendo grande flessibilità necessità di una maggior mole di calcoli. Nella valutazione degli indici XML che vedremo nella prossima sezione si tiene conto di quale tipo di query viene risposto con precisione da un indice.

Tabella 3.1: *Axis Specifiers.*

Full Syntax	Abbreviation	
ancestor		
ancestor-or-self		
attribute	@xyz	short for /attribute::xyz
child	/xyz	short for /child:xyz
descendant		
descendant-or-self	//	short for /descendant-or-self::node()
following		
following-sibling		
namespace		
parent	/..	short for /parent::node()
preceding		
preceding-sibling		
self	/.	short for /self::node()

3.3.1 XPath e Simulazione

Ramanan propone invece una divisione dei tipi di query più formale, definendo vari sottoinsiemi del linguaggio XPath, per mettere in luce l'importanza della relazione di simulazione nel contesto dell'elaborazione di dati XML.

Introduciamo CoreXPath (CXPath) come il linguaggio XPath a meno degli operatori aritmetici e sulle stringhe, e degli assi namespace e attribute che sostituiamo con idref e reverse-idref ; avremo dunque tredici assi, i predicati e gli operatori booleani or,and e not. La struttura di una query CXPath (e conseguente mente quella dei suoi sottoinsiemi) è così schematizzata:

Grammatica CXPath

```
<CXPath Query>    := <Absolute Query>
                   | <Relative Query>
<Absolute Query>  := / <Relative Query>
<Relative Query>  := <Location Step>
                   | <Location Step> / <Relative Query>
<Location Step>   := axis :: nt <Predicates>
<Predicates>      := ϵ | [<Predicate>]
                   | <Predicates>
<Predicate>       := <Predicate> and <Predicate>
                   | <Predicate> or <Predicate>
                   | or <Predicate>
                   | <Relative Query>
```

Definiamo Branching Path Queries il sottoinsieme di CXPath dove si ignorano gli assi relativi all'ordine, ovvero `preciding`,`preciding-sibling`,`following`, `following-sibling`.
Tree Pattern Queries è infine un sottoinsieme di BPQ dove sono consentiti solo i quattro assi `self`,`child`,`descendant`,`descendant-or-self` e l'operatore `and`; in particolare non si considerano gli archi idref. Avremo dunque:

$$XPath \supset CXPath \supset BPQ \supset TPQ$$

Si definiscono inoltre per ognuna delle classi appena descritte le rispettive versioni senza l'operatore booleano `not`. Data una classe C indicheremo la classe ottenuta come C^+. Possiamo scrivere:

$$XPath^+ \supset CXPath^+ \supset BPQ^+ \supset TPQ^+ = TPQ$$

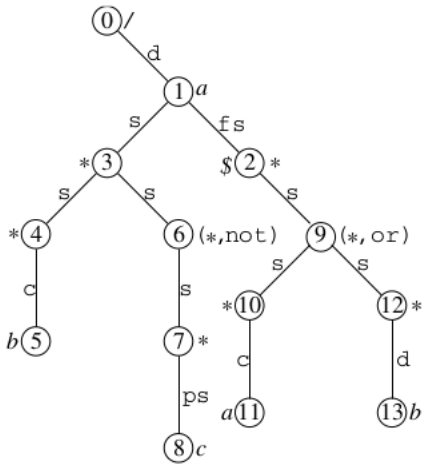


Figura 3.1: *Albero della query Q*

Partendo da una query CXPath o un suo sottoinsieme è possibile costruire un albero Q della query; in figura è mostrato l'albero che corrisponde alla query `Q = /d::`a`[c::`b `and not ps::`c`]/fs::*[c::`a `or d::`b`]` dove con d si intende l'asse descendant::, con c l'asse child, con ps l'asse preceding-sibling, con fs l'asse following-sibling.
Nell'albero della query ogni nodo ha un etichetta con il tipo e l'eventuale operatore `bool`. A ogni arco è associato un asse.
Per prima cosa si costruisce un percorso lineare, detto trunk(Q) ignorando tutti i predicati dalla query. ogni vertice di trunk(Q) corrisponde a un location step della query. L'ultimo nodo è contrassegnato con $ è corrisponde al

nodo di output della query.
I predicati possono essere ora aggiunti in maniera ricorsiva aggiungendo un arco con asse `self::` collegato al sotto albero del predicato; questo sottoalbero ha come radice un nodo di tipo * con l'eventuale operazione booleana (`and` può essere sottointesa). Viene creato un ramo per ogni operando che viene valutato ricorsivamente.

Ramanan dimostra due importanti risultati:

- Valutare un espressione di $CXPath^+$ equivale a calcolare la simulazione del grafo della query Q rispetto al grafo del documento D
- L'indice basato sulla simulazione è il più piccolo indice in grado di coprire le query di tipo BPQ^+

Il secondo risultato è interessante in quanto è noto che gli indici basati su bisimulazione preservano il linguaggio delle BPQ ma nel caso in cui si incontrano query che non fanno uso dell'operatore not è possibile sfruttare la riduzione spaziale offerta dagli indici basati su simulazione.

Capitolo 4

XML Indexing

Elaborare una query XPath significa visitare il documento e restituire i nodi che rispondo alle proprietà desiderate; nella visualizzazione ad albero del DOM questo corrisponde a una visita in profondità. Per documenti molto grandi tuttavia questa visita diventa molto onerosa anche per query molto semplici. Lo scopo degli indici XML è quello di semplificare l'operazione riducendo lo spazio e il tempo di ricerca in qualche modo.
Da un punto di vista formale Xpath è un linguaggio con una certa espressività in grado di distinguere tutti gli elementi di un documento ma una query molto spesso non utilizza a pieno tutto il potere espressivo a disposizione. Avremo allora delle classi di query per le quali alcuni nodi saranno indistinguibili. Un indice non è altro che una struttura dati in cui i nodi non distinguibili vengono accorpati, fornendo così una struttura dati semplificata su cui valutare le query. Va notato che a volte può essere utile utilizzare un indice anche se la risposta alla query risulterà non accurata (avremo dei falsi positivi) per poi effettuare un filtraggio dei risultati.

In questa sezione vedremo alcuni tipi di indici strutturali per documenti XML basati sui nodi e sui grafi.

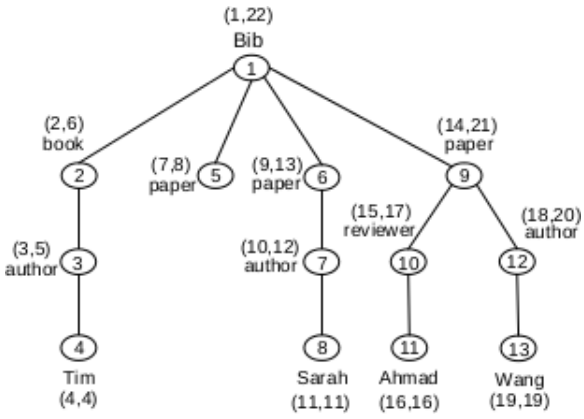
4.1 Indici Basati sui Nodi

Inteval e Path Labeling sono techiche di indicizzazione basate sui nodi. Immagazzinando dei valori che riflettono in qualche modo la posizione dei nodi nella struttura dell'albero XML di trovare, dato un nodo, il nodo padre, i figli, i fratelli, i discendenti e gli antenati. Mostriamo due varianti dette *Interval(o Region) Labeling* e *il Prefix(o Path) Labeling* che differiscono nel etichettamento.

4.1.1 Interval Labeling

Inteval Labeling significa etichettare i nodi con un intervallo in base alla posizione nell'albero o nel documento. Esempi di questo metodo sono la tecnica *(Pre,Post)* e la tecnica *(Beg,End)* che vedremo in dettaglio.
Nel metodo *(Pre,Post)* visitiamo l'albero XML con un algoritmo di visita Pre-Order e Post-Order. Ogni volta che incontriamo un nodo lo numeriamo con la posizione di visita. Confrontando le etichette possiamo ricostruire la posizione di ogni nodo nell'albero senza mantenere la struttura dati in memoria.
Nel metodo *(Beg,End)* scorriamo il documento XML in maniera sequenziale mantenendo un contatore che viene incrementato ogni qual volta incontriamo un tag, un attributo, o i valori a essi associati. Assegniamo il valore del contatore all'etichetta *Beg* dell'elemento quando lo incontriamo e quando giungiamo alla terminazione di questo assegniamo il valore del contatore a *End*.

Esempio 11



(a)

Label (Tag)	Beg	End	Level	Flag (Type)	Value
book	2	6	2	Element	Null
author	3	5	3	Value	Tim
paper	7	8	2	Element	Null
...	...	...	...	...	...
paper	14	21	2	Element	Null
reviewer	15	17	3	Attribute	Ahmad

(b)

Figura 4.1: *esempio di labeling (beg,end)*

Aggiungendo una terza etichetta *Level* che indica la profondità nell'albero possiamo formulare due regole per calcolare le relazioni successore-discendente e padre-figlio:

- **Proprietà 1:** in un albero un nodo x è antenato di un nodo y sse $x.Beg < y.Beg < x.End$
- **Proprietà 2:** in un albero un nodo x è padre di un nodo y sse $x.Beg < y.Beg < x.End$ and $y.Level = x.Level + 1$

4.1.2 Path Labeling

Questa tecnica si basa sull'etichettare ogni nodo con un vettore in cui è indicato il percorso fino ad esso. Come esempio di questa tecnica illustreremo il metodo Dewey.
Il metodo Dewey prevede che ogni etichetta rappresenti la posizione del nodo includendo come prefisso la codifica dei suoi antenati (coordinata verticale) e aggiungendo il numero del nodo tra i suoi fratelli (coordinata orizzontale). Il livello è implicitamente definito dalla lunghezza del vettore. Per stabilire le relazioni tra due nodi sarà sufficiente effettuare un controllo di pattern matching tra le etichette.
Gli antenati di un certo nodo x saranno tutti quelli la cui etichetta è una sotto-stringa dell'etichetta di x, il padre sarà il nodo la cui etichetta è soltanto un carattere più breve; ad esempio è facile verificare che il nodo (0.3) è antenato del nodo (0.3.1.0) e padre di (0.3.1).
I fratelli invece avranno un etichetta della stessa lunghezza che differirà soltanto per l'ultimo carattere; ad esempio (0.3.1) e (0.3.2).
Il punto di forza di questo sistema è la semplicità nella verifica delle relazione e nell'aggiornamento della struttura; la grande debolezza invece la lunghezza delle etichette cresce con l'aumentare delle profondità e con essa lo spazio necessario ad immagazzinare le informazioni e il tempo necessario a calcolare le relazioni tra i nodi.

4.2 Strong DataGuide

Proposto nel 1997 da Goldman e Widom, è uno dei primi algoritmi utilizzati per il calcolo degli indici. Partendo da un grafo G_1, Strong DataGuide restituisce un grafo G_2 in cui i nodi vengono partizionati in base al percorso dalla radice al nodo. Pensando il grafo di partenza come un automa a stati finiti **non-deterministico** l'algoritmo genera il grafo dell'automa a stati finiti **deterministico** corrispondente.
Un indice Strong DataGuide deve soddisfare le seguenti caratteristiche di base:

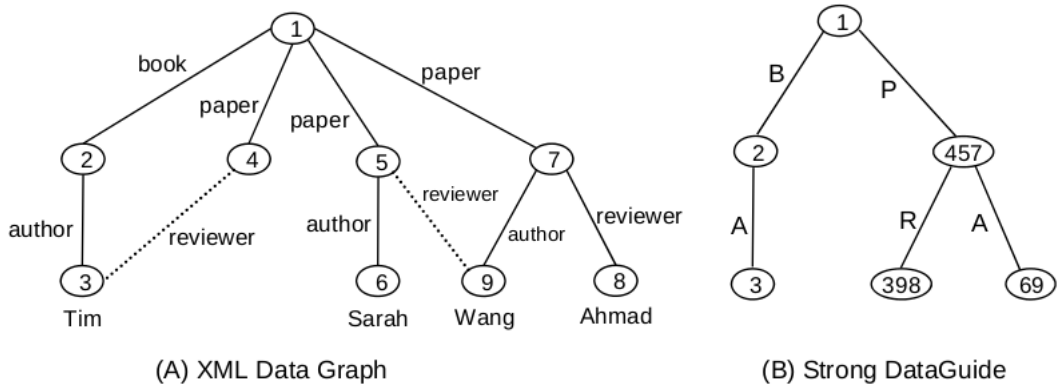
- Ogni cammino distinto radice-nodo nel grafo di partenza compare soltanto una volta nell'indice.

- Ogni singolo cammino nel grafo indice deve avere almeno una corrispondenza nel grafo originale, ossia non ci sono percorsi non validi nell'indice.

Essere deterministico è contemporaneamente un vantaggio e uno svantaggio in quanto i nodi con più di un padre(idref) verranno duplicati. Se consideriamo un documento con idref modellato quindi come un DAG esiste la possibilità che la cardinalità dell'indice sia maggiore di quella del grafo di partenza, vanificando lo scopo dello stesso. Nel caso invece di un documento modellato come un albero nel caso peggiore la cardinalità dell'indice sarà uguale a quella del documento.

DataGuide è in grado di fornire risposte precise e complete alle cosìdette path-query, cioè quelle query che presentano sono relazioni padre-figlio o antenato-discendente. Nel caso di query che coinvolgono anche i predicati (twig-query) la risposta ottenuta sarà completa ma non precisa e otterremo dei falsi positivi.

Esempio 12



4.3 Indici Basati sulla Bisimulazione

Vediamo due tipi di indici basati sulla simulazione: A(1)-index, in grado di coprire Path Queries, e F&B-index, in grado di coprire anche le Twig Queries. Entrambi questi indici risultano poco efficienti nel caso in cui i dati siano molto grandi e irregolari per via dei requisiti stringenti della relazione di bisimulazione

4.3.1 1-index

Introdotto nell 1999 da Milo e Suciu questo indice è basato sulla relazione di Bisimulazione all'indietro (Backward Bisimiluation), questa relazione è

uguale alla bisimulazione vista in precedenza con la differenza che conserva gli archi entranti invece di quelli uscenti. Di seguito una definizione formale.

Definizione 10 (Backward Bisimulation)**.** *Dati due vertici u e v, $u \simeq v$ se:*

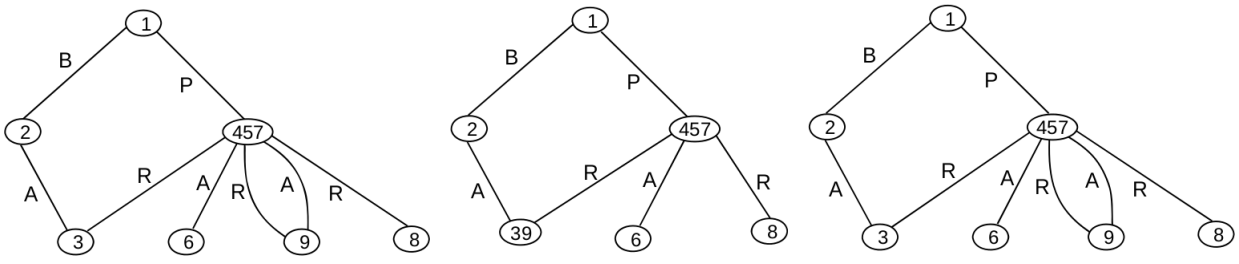
- $\langle\langle u \rangle\rangle = \langle\langle v \rangle\rangle$
- $\forall u' \in pre(u) \exists v' \in pre(v) | u' \simeq v'$
- $\forall v' \in pre(v) \exists u' \in pre(u) | v' \simeq u'$

Diremo quindi che u B-bisimula v; se $u \simeq v$ e $v \simeq u$ allora si ha una relazione di equivalenza che chiameremo backward-bisimulazione $\approx^B$*.*

1-index non è altro che il partizionamento del grafo di un documento XML secondo le classi di equivalenza della relazione di B-Bisimulazione. Per ottenere l'indice si può procedere invertendo gli archi del grafo e utilizzando un algoritmo che applica la definizione di F-bisimulazione vista nel capitolo introduttivo.

Esistono anche delle versioni approssimate dell'algoritmo 1-index, dette A(k)-Index e D(k)-index. A(k)-index calcola la bisimulazione su percorsi entranti di lunghezza k, con k impostato manualmente, invece che fino alla radice, diminuendo la precisione ma aumentando la riduzione dello spazio.D(k)-index opera alla stessa maniera ma il valore di k viene impostato dinamicamente.

Esempio 13



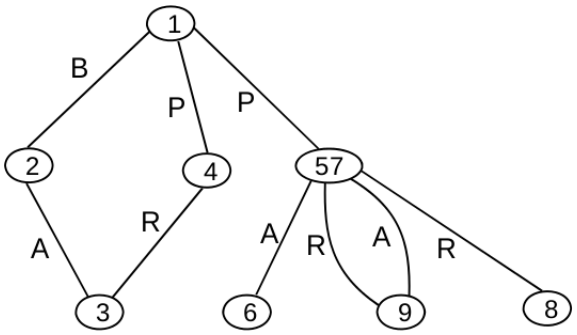
4.3.2 F&B-index

Sviluppato nel 2002 da Abiteboul, Buneman, et al. F&B-index (Forward & Backward Index) è un indice basato su bisimulazione in avanti e in indietro.

Questa relazione è l'unione delle condizione di Forward-simulation e Backward-simulation e permette di conservare i tipi e i gli archi sia entranti che uscenti. A differenza degli indici visti in precedenza questo tipo indice è in grado di rispondere accuratamente anche a query ramificate. Kaushik, Bohannon, Naughton, e Korth (2002) hanno dimostrato che F&B-index è l'indice più piccolo in grado di coprire le Branching Path Queries. Questo guadagno in versatilità viene però a discapito delle dimensioni dell'indice che sarà più grande per via della relazione d'equivalenza più esclusiva.

Come per gli indici basati su B-bisimulazione anche per questo indice sono state proposte varianti approssimate, in particolare $(F\&B)^k$-index dove k è il parametro che gestisce l'approssimazione,e quindi le dimensioni, dell'indice in maniera affine a A(k)-index. Un altra variante dell'indice è pensata per distribuire i risultati su disco in modo da non riempire la memoria centrale, consentendo così di lavorare con documenti più grandi.

Esempio 14



(F) F&B

4.4 Indici Basati sulla Simulazione

Capitolo 5

Implementazione