XML Covering Index utilizzando la relazione di Simulazione

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

1 aprile 2015

Indice

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.

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 originariamente 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 pre-

cedenti 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.

2 Struttura, Sintassi e Semantica dei Documenti XML

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 1. Ecco un esempio di codice XML

```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>
```

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 2. 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 3. 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>
            <xs:sequence>
              <xs:element type="xs:string" name="sub-element"
                      maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute type="xs:byte" name="attribute2"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

3 Interrogazione e manipolazione di dati XML

All'interno di applicazioni i dati XML vengono rappresentati in forma di DOM e manipolati tramite API specifiche per il linguaggio che viene usato. La struttura ad Albero del DOM 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 essere acceduti da remoto o provenire da un'altra applicazione. É emersa allora la necessità di sviluppare un linguaggio atto all'esplorazione e l'interrogazione dei file XML. Di seguito descriviamo tre linguaggi importanti: *XPath*, *XQuery* e *XSLT*.

3.1 Il Linguaggio 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 complessi 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 `/`, all'interno di un elemento possiamo trovare 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 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`.

4 Memorizzare dati XML

Scardina, Chang and Wang (2004) offrono tre opzioni per l'archiviazione di dati XML. Queste sono salvare il documento XML com'è, estrarre i dati dal

Tabella 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()

documento e mantenere i dati in un database relazionale, oppure usare database che possiedono un tipo di dato nativo per XML.Se pensiamo a un sistema di gestione di contenuti, che lavora su documenti XML allora non è difficile immaginare di lasciare i file XML come sono, senza bisogno di processarli. Oracle e altri RDBMS offrono tipi come large character che possono supportare un documento XML memorizzato come una lunga stringa. Alternativamente, si potrebbe semplicemente memorizzare i file XML sul file system. Il risultato è praticamente lo stesso, tutto ciò che si può fare con i documenti è leggerli o scriverli nella loro interezza.
Se questo approccio è adatto all'applicazione, allora lo sviluppatore dovrebbe pensare a comprimere il file XML prima di salvarlo. La sintassi di XML è molto verbosa, e spesso può far aumentare significativamente lo spazio richiesto dal file nonostante la quantità di informazione contenuta. Il *W3C Efficient XML Interchange Working Group* ha la responsibilità di trovare modi per risolvere la verbosità di XML, e fornire un'efficiente mezzo per comprimere XML per il trasporto e la memorizzazione (Cover, 2007).Anche Harrusi, Averbuch, Yehudai (2006), hanno pubblicato una tecnica di compressione XML-aware.
Dal momento che molte applicazioni sono focalizzate sui dati e i contenuti del documento XML il primo approccio non è utilizzabile. La seconda opzione è fare il parsing dei dati dal documento XML e salvarli come normali dati relazionali. Poiché XML dovrebbe essere auto-descrittivo, nel senso che contiene sia i dati che il loro significato, dovrebbe essere possibile scorrere il

file XML pero trovare nei dati i punti di interesse per l'applicazione, e salvarli in una tabella relazionale o relazionale a oggetti(?).
Una conseguenza di questo approccio è che è richiesta una quantità significativa di calcoli nel parsing di XML,tanto più quando si lavora con grandi dataset. Lu, Chiu, Pan (2006) hanno proposto una approccio parallelo per questi casi, distribuendo il carico computazionale tra varie CPU.
Infine Scardina, Chang and Wang (2004) raccomandano di memorizzare i dati XML come tipi XML nativi nei database. Tutto ciò che è necessario in questo approccio è registrare lo schema XML sul database Oracle. Questo processo di registrazione permette al database di 'conoscere' quali tipi di dato si sta memorizzando, e d ci creare lo spazio appropriato. *Oracle's XML developer kit* (XDK) fornisce un ricco set di *application programming interfaces* (APIs) per occuparsi dei dati in questo contesto.
Questa tecnica restituisce risultati migliori di entrambe le precedenti soluzioni con facilità di memorizzazione, e accesso in stile relazionale ai dati contenuti usando query SQL. Un approccio del genere è indicato solo nei casi in cui è definito uno schema XML, in caso contrario o nel caso in cui lo schema subisca frequenti modifiche la soluzione perde di efficienza.

Simulazione e Grafo Quoziente

In questa sezione introduciamo la relazione di *simulazione*, il concetto di *grafo quoziente*, i campi in cui questi argomenti vengono utilizzati e infine un algoritmo per il calcolo della relazione. La relazione di simulazione e le altre relazioni di d'equivalenza ad essa collegate possono essere utilizzate per ridurre la cardinalità di un grafo preservandone la capacità espressive, il grafo ridotto è detto grafo quoziente e risulta molto utile nel model checking. Per calcolare le classi d'equivalenza che servono a costruire il grafo quoziente ci avvaliamo dell'algoritmo HHK illustrato nell'ultima parte.

1 Relazione di Simulazione

Definizione 1. *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* $\langle\langle v \rangle\rangle$.

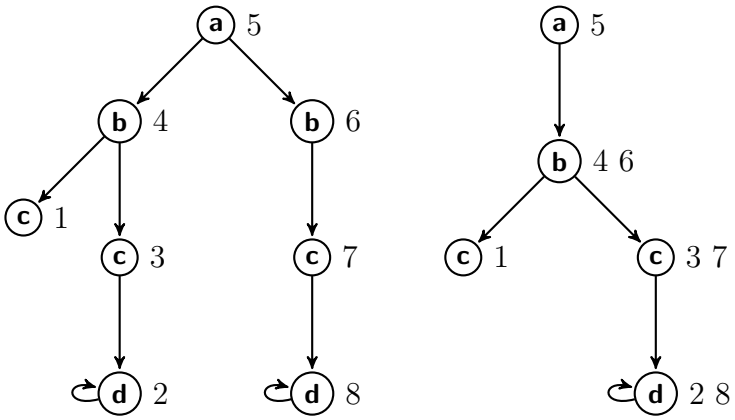
Data la funzione $post(v) = \{u | (v, u) \in E\}$ *che restituisce l'insieme dei successori del vertice* v. *Possiamo definire una relazione binaria* $\preceq \subseteq V^2$ *detta* simulazione *se si verificano le seguenti condizioni:*

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.*

1.1 Utilizzi

Le classi di equivalenza formate dalla relazione $\approx^S$ possono essere usate per costruire un nuovo grafo, detto grafo quoziente. Il grafo quoziente è molto utilizzato nel Model Checking con lo scopo di semplificare la verifica di espressioni di logica temporale. Partendo da una struttura di Kripke infatti possiamo applicare la relazione di simulazione e ottenere una struttura ridotta che riconosce un sottoinsieme della logica ∀CTL*.



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

Figura 1: *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.*

Come vedremo in seguito questa tecnica può essere applicata agli indici XML; infatti il DOM di un documento XML è un albero, ossia un grafo particolare, e le Tree Pattern Query che utilizziamo per ottenere informazioni possono essere assimilate alla logica di primo grado, come la LTL, un sottoinsieme di CTL*.

2 Altre Relazioni di Riduzione

Vi sono altre due relazioni d'equivalenza applicabili ai grafi connesse alla similarità: La trace equivalence $\approx^T$ e la bisimulazione $\approx^B$. La prima è una relazione più grossolana della similarità, mentre la seconda è una relazione più fine.

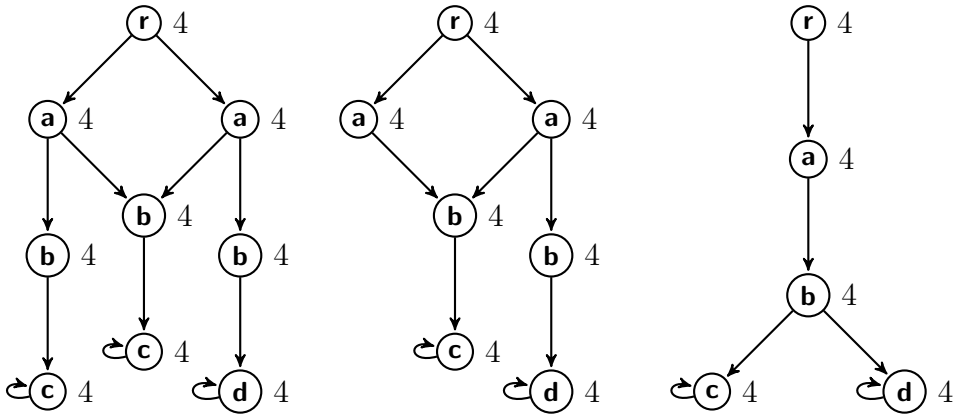
2.1 Trace Equivalence

Definizione 2. *Dati due vertici u e v, diciamo che u domina v se:*

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

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.*



(a) Grafo G **(b)** Bisim G **(c)** TraceEq G

Figura 2: *Esempi di Trace Equivalence e Bisimulazione. A sinistra abbiamo un grafo G di partenza, al centro il grafo quoziente simG e a destra il grafo quoziente TraceG. Notare come la Bisimulazione non ha grandi capacità di riduzione rispetto a trace equivalence e simulazione.*

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.

2.2 Bisimulazione

Definizione 3. *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 $O(mlogn)$.

Figura 3: *Esempio di labeling (Beg,End)*

3 Algoritmo HHK per il calcolo della Simulazione

Negli anni sono stati propost*(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*. 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 $y.Level = x.Level + 1$

i 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.

3.1 Descrizione

L'algoritmo 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 deterministica, 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.

Algorithm 1 Algoritmo 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: $remove(v) := \emptyset$

3.2 Correttezza e Complessità

L'algoritmo 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

Figura 4: *Algoritmo HHK in azione. Nella fase a...*

$\Sigma_v \Sigma_w (|pre(v)| = O(mn))$ 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).

due parole sulla correttezza...

3.3 Algoritmo in Esecuzione

Prendiamo in considerazione un grafo come quello in figura. All'inizio(fig.a) l'algoritmo genera gli insiemi dei simulatori per ogni nodo (righe n..m) otteniamo i due insiemi colorati. Successivamente, nel ciclo while inizia a controllare i predecessori di ogni nodo e loro simulatori affinando gli insiemi(fig.b). Scopriamo che i nodi non sono simili e quindi li rimuoviamo dagli insiemi tramite l'insieme remove (riga x). a fine algoritmo otterremo gli insiemi dei simulatori per ogni nodo, con queste classi di equivalenza possiamo costruire il nostro grafo quoziente (fig.c).

Indicizzazione Strutturale di dati XML

1 Cos'è un indice strutturale

Un indice strutturale sta a un database XML come uno schema sta a un database relazionale. Essi riflettono la struttura e le relazioni interne ai dati e vengono utilizzati per verificare la validità delle query ancora prima che vengano eseguite. Ad esempio possiamo controllare, sia in query lineari che ramificate, l'esistenza di un percorso nel DOM prima di procedere a qualsiasi tipo di elaborazione.
Generalmente possiamo dividere gli indici strutturali in tre categorie: **Node Indexes**, **Graph Indexes**, **Sequence Indexes**,

Indici di Nodi: questo metodo è basato sull'etichettare i nodi in vari modi come Interval labeling o prefix labeling. Entrambi sono particolarmente indicati per dati organizzati ad albero.

Indici di Grafi: questo metodo si basa sull'utilizzo di grafi. Gli indici di questo tipo coprono i percorsi lineari e ramificati oppure soltanto lineari.

Indici di Sequenze: questo metodo tratta la query come una sequenza da verificare contro i dati xml utilizzando tecniche di pattern matching.

2 Node Indexes

Gli indici di nodi immagazzinano dei valori che riflettono in qualche modo la posizione dei nodi nella struttura dell'albero XML. Possono essere utilizzati per trovare, dato un nodo, il nodo padre, i figli, i fratelli, i discendenti e gli antenati. Alla base di questo metodo di indicizzazione ci sono delle tecniche di labeling dei nodi, due delle più diffuse sono *Interval(o Region) Labeling* e *il Prefix(o Path) Labeling*.

2.1 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

Figura 5: *esempio di labeling (beg,end)*

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*.
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 $y.Level = x.Level + 1$

2.2 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.

3 indici grafo

4 indici sequenza