Università degli Studi "G.D'Annunzio" Chieti - Pescara

FACOLTÀ DI ECONOMIA
Corso di Laurea in Economia Informatica

Risoluzione efficiente di interrogazioni XPath su documenti XML con attributi e riferimenti

Tesi di Laurea in Algoritmi e Strutture di Dati

Relatore:
Dott. Massimo Franceschet

Laureando:
Enrico Zimuel

Anno Accademico 2004/5

Indice

Introduzione

Obiettivo di questa tesi di laurea è la realizzazione di un algoritmo efficiente per la risoluzione di interrogazioni *XPath* su documenti XML con attributi e riferimenti. Il linguaggio *XPath* è un linguaggio che consente di ricercare elementi all'interno di un documento XML.

Nel corso di questi ultimi anni sono stati proposti diversi algoritmi per la risoluzione di interrogazioni *XPath*, ma non tutti risultano essere efficienti in termini di velocità di esecuzione. Nell'articolo "*Efficient Algorithms for Processing XPath Queries*" [1] dei ricercatori Georg Gottlob, Christopher Koch e Reinhard Pichler e pubblicato nel 2002 negli atti della conferenza *Very Large DataBases 2002*, viene messa in evidenza l'inefficienza di alcune famose implementazioni di *XPath* come XALAN, XT e IE6, addirittura con fattori di complessità computazionale addirittura esponenziali.

Gli stessi Gottlob, Koch e Pichler dimostrano, nello stesso articolo, che sia possibile risolvere in maniera efficiente un sottoinsieme del linguaggio *XPath* con un fattore di complessità computazionale al più lineare.

L'idea che ha ispirato la presente tesi di laurea è stata quindi quella di provare ad ampliare i risultati ottenuti nell'articolo suddetto, su un sottoinsieme più ampio di *XPath*, che comprendesse documenti con attributi e riferimenti.

Oltre alla realizzazione teorica dell'algoritmo si è quindi pensato di implementarlo su di un calcolatore al fine di verificarne l'efficienza tramite alcuni test su documenti XML di diverse dimensioni.

Oltre ai test di efficienza, è stato poi eseguito un confronto con un processore di interrogazioni noto nella comunità scientifica e idoneo a valutare la bontà del lavoro svolto.

Più in dettaglio, le fasi che hanno portato alla realizzazione del presente lavoro di tesi hanno riguardato: lo studio del linguaggio XML con attributi e riferimenti; l'identificazione di un frammento di *XPath* per interrogare documenti con attributi e riferimenti, con specifica della sua sintassi e semantica; l'ideazione dell'algoritmo per la risoluzione di interrogazioni nel frammento *XPath* scelto ed analisi della complessità computazionale; implementazione in linguaggio C del processore di interrogazioni; test di efficienza e confronto con un processore efficiente noto nella comunità scientifica internazionale.

Capitolo 1

Uno sguardo a XML

1.1 Che cos'è XML?

XML (eXtensible Markup Language) è un linguaggio che consente di definire dati *semistrutturati*.

Un dato *semistrutturato* è una tipologia di dato che contiene al suo interno alcune informazioni sulla sua struttura oltre al dato in se. Spesso questo tipologia di dati viene anche definita *senza schema* (schemaless) o *auto-descrittiva* (self-describing) [2]. Tipicamente quando si memorizzano delle informazioni di solito se ne definisce prima la struttura, lo schema, e successivamente si inseriscono i valori, definiti anche *istanze*. Utilizzando dati semistrutturati è possibile unire le due operazioni precedenti tramite l'utilizzo di una semplice sintassi in grado di definire la struttura ed il valore dei dati contemporaneamente. Proprio per questa natura di sintesi informativa i documenti XML sono molto utilizzati per le applicazioni di trasferimento dati, soprattutto su Internet. Un altro grande vantaggio dei documenti XML è il fatto che essi sono scritti in puro formato testuale, un formato universalmente accettato da tutti i sistemi hardware e software.

L'eXtensible Markup Language è nato nel febbraio 1998 da una raccomandazione [3] del W3C (World Wide Web Consortium[1]), uno dei consorzi di aziende più importanti per la creazione di standard utilizzati su Internet.

XML è un linguaggio derivato dallo standard SGML (Standard Generalized Mark-

[1] http://www.w3c.org

up Language[2]), un metalinguaggio, ossia un linguaggio per la creazione di altri linguaggi, utilizzato per la creazione di Markup Language. XML è un sottonsieme molto snello dell'SGML. Secondo le specifiche [3] del consorzio W3C gli obiettivi più importanti che hanno spinto alla creazione dello standard XML sono stati:

- facilità di utilizzo su Internet;
- ampio supporto di applicazioni;
- compatibilità con l'SGML;
- facilità di implementazione di programmi che utilizzano documenti XML;
- facilità di lettura ed interpretazione dei documenti XML;
- facilità nella creazione di documenti XML;

Molto spesso l'XML viene confrontato con l'HTML (HyperText Markup Language), il linguaggio utilizzato per la creazione delle pagine web [4]. L'HTML è un linguaggio rigido utilizzato semplicemente per impaginare pagine web, l'XML è un linguaggio flessibile che consente di descrivere il contenuto di un documento, come ad esempio una pagina web. La differenza è fondamentale tanto che l'HTML può essere considerato come un sottoinsieme dell'XML. L'XML può essere utilizzato anche come *database* per la memorizzazione di grandi quantitativi di dati, anche se con alcune limitazioni [5, 6, 7]. Anche se l'XML non è stato progettato come strumento per la memorizzazione di grandi quantità di informazioni esistono diverse applicazioni che utilizzano con successo *database* nativi XML [8, 9].

1.2 La sintassi XML

Un documento XML è una rappresentazione testuale di informazioni identificate per mezzo di *tag*, ossia di stringhe racchiuse tra i simboli < e >. Attraverso l'utilizzo dei tag è possibile definire la struttura di un documento XML. Il componente fondamentale di un documento XML è l'*elemento*, una stringa costituita da un *tag di apertura*, un valore, eventualmente vuoto, e un *tag di chiusura* (ad esempio

[2]Standard ISO 8879 del 1986

<name>Alan</name> è la definizione di un elemento `name` che contiene il valore `Alan`). Il contenuto di un tag identifica in qualche modo la tipologia del dato. Nell'esempio precedente il tag <name> ci fornisce un'informazione chiara e sintetica sulla sua semantica, ad esempio in questo caso risulta evidente che si stà definendo il nome di una persona.

I tag di apertura e quelli di chiusura vengono anche chiamati *markup*[3].

Attraverso l'utilizzo dei tag si possono definire delle strutture dati di tipo gerarchico dove gli elementi sono costituiti da più elementi figli che a loro volta possono essere costituiti da altri elementi, figli dei figli, e così via. Ad esempio in Figura 1.1 è riportato un elemento <person> definito da 3 elementi figli <name>, <surname> e <age>.

```
<person>
    <name>Alan</name>
    <surname>Turing</surname>
    <age>42</age>
</person>
```

Figura 1.1: Esempio di elemento XML di tipo gerarchico

I tag possono essere di qualsiasi tipo e possono essere scelti liberamente da chi realizza il documento XML a patto di rispettare alcune regole: devono iniziare con un carattere o con un *underscore* (_); non possono inziare con numeri; possono contenere un qualsiasi numero di lettere, numeri, trattini, punti ed *underscore*; non possono contenere spazi; sono *case-sensitive* quindi, ad esempio, <Name> e <name> sono elementi differenti; conviene evitare di utilizzare tag riservati del linguaggio, come ad esempio <xml>.

Gli *elementi* di un documento XML possono contenere anche degli *attributi* ossia dei valori racchiusi tra apici o doppi apici definiti all'interno dello stesso *tag* dell'elemento, ad esempio <age unit='years'>42</age> definisce l'attributo `unit`, dell'elemento <age>, con valore pari a `'years'`. Gli *attributi* vengono utilizzati per definire proprietà specifiche di un *elemento*, nell'esempio precedente l'attributo `unit` specifica l'unità di misura dell'età della persona, espressa in anni (*years*). I nomi

[3]In questa tesi si utilizzerà la parola inglese *markup* al posto della sua traduzione italiana *marcatore*.

degli attributi possono essere scelti liberamente a patto di rispettare le stesse regole di nomenclatura dei tag viste in precedenza. Inoltre ogni attributo può comparire una sola volta all'interno dello stesso tag, non è quindi possibile avere un tag che presenti due volte lo stesso attributo (questo per evitare ambiguità in fase di lettura del documento XML). Esistono degli attributi riservati che non possono essere utilizzati liberamente ma solo con il significato che è stato loro attribuito.

Alcuni di questi attributi riservati sono:

- xml:lang= it | en | ..., indica la lingua del tag corrente;
- xml:space= preserve | default, indica se gli spazi del contenuto testuale del tag devono essere considerati (*preserve*) oppure possono essere considerati superflui (*default*).
- xml:attribute= "...": permette di rinominare gli attributi per evitare conflitti in fase di lettura o di utilizzo di altre tecnologie XML.

Esistono poi alcuni caratteri che non possono essere inseriti come valori testuali così come sono all'interno di un documento XML, ad esempio i caratteri < e > non possono essere inseriti come valori poichè verrebbero confusi con la definizione di un elemento XML.

Per evitare questo possibile conflitto è necessario utilizzate un codice, denominato *entity-name*, al posto del carattere che si desidera inserire, ad esempio in Figura 1.2 sono riportati alcuni *entity-name* di alcuni caratteri particolari.

Entity-name	**Carattere**
&amp;	&
&lt;	<
&gt;	>
&quot;	"
&apos;	'
&#x??;	Chr(??)

Figura 1.2: Alcuni *entity-name* XML.

L'ultimo *entity-name* della tabella di Figura 1.2 consente di rappresentare qualsiasi caratteri a partire dal suo codice ASCII, espresso in esadecimale, al posto di ??.

In un documento XML possono essere inseriti dei commenti tramite il tag di apertura <!- - e tramite il tag di chiusura - ->. In questo modo il testo compreso tra <!- - e - -> non verrà considerato durante la lettura del documento XML.

Una regola fondamentale per la creazione di documenti XML è quella relativa al prologo (*header*) del documento. Ogni documento XML deve iniziare con il seguente tag particolare:

<?xml version=”versione” enconding=”codifica” standalone=”yes/no”?>

Questo tag viene chiamato *processing instruction* e serve per indicare che si tratta di un documento XML nella versione specificata, che utilizza la codifica specificata e se ha o meno una definizione di struttura (*standalone*) autonoma o esterna.

Un'altra regola fondamentale è quella dell'unicità dell'elemento di primo livello; ossia ogni documento XML deve avere un solo elemento che racchiude tutti gli altri elementi XML.

1.3 Documenti XML ben formati

Un documento XML di dice ben formato (*well formed*) quando rispetta le regole sintattiche definite in precedenza. Un documento XML deve essere ben formato per poter essere utilizzato da un software che dovrà leggerlo ed interpretarlo[4] correttamente. Ad esempio il seguente documento XML non è ben formato:

```
<?xml version="1.0"?>
<person>
  <Name>Alan</name>
  <surname>Turing</surname>
  <age 1unit="years">42</age>
</person>
```

poichè il tag di apertura `<Name>` è diverso dal tag di chiusura `</name>` ed ancora perchè la definizione dell'attributo `1unit` non può iniziare con un numero.

[4]I software che leggono ed interpretano i documenti XML si dicono *parser* in quanto analizzano pezzo per pezzo la struttura del documento.

1.4 Document Type Definition (DTD)

Fino a questo momento abbiamo parlato solo di *sintassi* XML e abbiamo visto come realizzare un documento XML ben formato, ora discuteremo della possibilità di associare una struttura ad un documento XML e quindi in un certo senso della possibilità di associare una *semantica* ad un documento XML.

Il processo con il quale viene verificata la correttezza sintattica (leggibilità) e la correttezza strutturale di un documento XML viene denominato *validazione*.

Per definire la struttura di un documento XML si possono utilizzare fondamentalmente due tecniche:

- DTD (Document Type Definition) si tratta della prima tecnologia utilizzata per definire la struttura di un documento XML. E' supportata praticamente da tutti i parser XML.
- XSD (Xml Schema Definition) è una raccomandazione del 2001 del W3C che rappresenta ormai uno standard per la validazione XML. Tutti i produttori di *parser* si stanno orientando verso questo tipo di tecnologia.

In questa tesi discuteremo solo dello standard DTD analizzando soltanto un sottoinsieme della sua sintassi anche perchè il processo di *validazione* di un documento XML è secondario rispetto alle finalità della tesi.

Le DTD possono essere di due tipi:

- Pubbliche: disponibili liberamente e consultabili da chiunque. Su Internet si possono trovare degli archivi (*repository*) contenenti DTD standard, ad esempio HTML è definito tramite DTD pubbliche depositate presso il W3C.
- Di sistema: riservate. Sono riservate solo per un utilizzo interno del sistema e non possono essere consultate liberamente.

Per poter associare uno schema DTD ad un documento XML si possono utilizzare due alternative: possiamo inserire la DTD come parte integrante del documento (DTD interna) oppure possiamo utilizzare una definizione esterna, di sistema o pubblica (DTD esterna).

Nel caso di DTD interna la sintassi da utilizzare è la seguente:

```
<?xml version="1.0" ... ?>
<!DOCTYPE nome [contenuto della DTD]>
```

Nel caso di DTD esterna di sistema la sintassi risulta essere:

```
<?xml version="1.0" ... ?>
<!DOCTYPE nome SYSTEM "URL del file DTD">
```

e nel caso di DTD esterna pubblica:

```
<?xml version="1.0" ... ?>
<!DOCTYPE nome PUBLIC "URL del file DTD">
```

Con uno schema DTD è possibile definire la struttura di un documento XML a partire dalla composizione degli *elementi* e degli *attributi*. Per definire la struttura di un elemento XML all'interno di una DTD è necessario utilizzare la parola chiave `ELEMENT` attraverso il seguente tag:

```
<!ELEMENT nome regola_DTD>
```

dove `nome` è il nome dell'elemento XML (il *tag*), la `regola_DTD` indica invece il tipo di contenuto che questo *tag* avrà ed eventualmente la sua relazione con altri contenuti descritti nella DTD. I valori che possono essere assegnati ad una `regola_DTD` sono:

- ANY: indica che il contenuto del *tag* è libero.
- PCDATA (Parsed Character Data): indica che il contenuto è solo ed unicamente di tipo testo.
- EMPTY: indica che il *tag* sarà costituito solo da attributi e non conterrà altri elementi al suo interno.
- Gruppi di elementi: indica che l'elemento è costituito da altri elementi.

Prima di procedere oltre con la definizione della sintassi DTD analizziamo il seguente documento XML che rappresenta una semplice base dati di una banca definita attraverso una DTD di sistema memorizzata nel file `bank.dtd`:

```
<?xml version="1.0"?>
<!DOCTYPE bank SYSTEM "bank.dtd">
<bank>

  <customer customer-id="C1" accounts="A1 A2" type="old">
    <name>Alan</name>
    <surname>Turing</surname>
  </customer>

  <customer customer-id="C2" accounts="A2" type="new">
    <name>Isaac</name>
    <surname>Newton</surname>
  </customer>

  <account account-number="A1" owners="C1">
    <branch-name>Londra</branch-name>
    <balance>1500</balance>
  </account>

  <account account-number="A2" owners="C1 C2">
      <branch-name>Cambridge</branch-name>
      <balance>2500</balance>
  </account>

</bank>
```

Ad esempio una possibile definizione DTD dell'elemento `name` può essere:

```
<!ELEMENT name (#PCDATA)>
```

Come ad esempio una possibile definizione DTD dell'elemento `customer`:

```
<!ELEMENT customer (name,surname)>
```

In quest'ultimo caso definiamo l'elemento `customer` composto dagli elementi `name` e `surname`.

Nella definizione di gruppi di elementi può essere utilizzato il carattere | per indicare un'alternativa, ad esempio la definizione `<!ELEMENT customer (name | surname)>` indica che l'elemento `customer` può essere costituito da un elemento `name` o da un elemento `surname` ma non da entrambi o da nessuno dei due.

Qualora sia necessario indicare una molteplicità di elementi si possono utilizzare i seguenti simboli:

- **?** che indica zero o 1;
- + che indica da 1 a infinite volte;
- * che indica da zero a infinite volte.

Ad esempio per indicare che l'elemento `bank` è costituito da un numero arbitrario, eventualmente nullo, di elementi `customer` ed `account` si utilizza la seguente definizione DTD:

```
<!ELEMENT bank (customer*, account*)>
```

Fino a questo momento abbiamo visto come definire la struttura di un elemento XML ora analizziamo della definizione della struttura degli attributi.

Per poter definire la struttura di un attributo è necessario utilizzare la seguente sintassi DTD:

```
<!ATTLIST elemento nome_attributo tipo valore_default>
```

dove `elemento` è il nome del tag del quale stiamo definendo l'attributo, il `nome_attributo` è l'attributo, `tipo` è la specifica della definizione del tipo di attributo e `valore_default` è l'eventuale valore predefinito.

In Figura 1.3 sono riportati alcuni tipi di attributi che possono essere specificati all'interno di una definizione DTD.

Tipo attributo	**Descrizione**
CDATA	Dati di tipo carattere
ENTITY	Il valore dell'attributo deve fare riferimento ad un'entità esterna dichiarata nella DTD
ENTITIES	Così come per ENTITY ma con la possibilità di specificare più valori separati da spazi
ID	Identificatore univoco di un elemento, utile per la definizione di relazioni
IDREF	Puntatore ad un ID univoco del documento XML o di un documento XML esterno
IDREFS	Così come per IDREF ma con la possibilità di specificare più valori separati da spazi

Figura 1.3: Alcuni tipi di attributi utilizzati negli schemi DTD

Quando si definiscono gli attributi per gli elementi è anche possibile dichiarare dei vincoli relativi alla presenza o meno degli stessi tag. Ad esempio si possono specificare i seguenti vincoli:

- #IMPLIED indica che il valore dell'attributo non è obbligatorio;
- #REQUIRED indica che il valore dell'attributo è obbligatorio;
- #FIXED indica che il valore dell'attributo è costante.

Per riassumere la sintassi DTD finora esposta riportiamo, di seguito, un esempio completo di DTD, il file `bank.dtd`, contenente lo schema DTD dell'esempio XML precedente della banca.

```
<!ELEMENT bank (customer*, account*)>
<!ELEMENT customer (name,surname)>
<!ATTLIST customer customer-id ID #REQUIRED accounts IDREFS #REQUIRED
type CDATA "old">
<!ELEMENT account (branch-name, balance)>
<!ATTLIST account account-number ID #REQUIRED owners IDREFS #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT branch-name (#PCDATA)>
<!ELEMENT balance (#PCDATA)>
```

1.5 Documenti XML con attributi e riferimenti

In questa tesi ci concentreremo su una tipologia particolare di documenti XML, quelli che contengono al loro interno *attributi* e *riferimenti* (ossia attributi di tipo ID, IDREF e/o IDREFS specificiati secondo una schema DTD).

In generale un documento XML può essere rappresentato tramite una struttura dati ad albero, in particolare un albero radicato etichettato [2]. I nodi dell'albero possono essere fondamentalmente di tre tipi: *elementi*, *attributi* e *valori testuali*.

Un documento XML che contiene al suo interno *riferimenti* e quindi attributi di tipo ID ed attributi di tipo IDREF e/o IDREFS può essere rappresentato con una struttura dati a *grafo* e non più ad *albero* poichè, di fatto, gli attributi di tipo IDREF e IDREFS non sono altro che *puntatori* agli elementi che contengono attributi di tipo ID (questo aspetto risulterà più chiaro quando ci occuperemo della funzione id() del linguaggio XPath[5]).

Questa nuova struttura dati che emerge dalla definizione di un documento XML con attributi e riferimenti risulta essere quindi più complicata di una semplice struttura ad albero.

In questa tesi, indagheremo la complessità di alcune tecniche di navigazione all'interno di questo struttura a grafo, tramite l'implementazione di un sottoinsieme del linguaggio XPath che verrà presentato nel prossimo Capitolo.

In particolare dimostreremo che è possibile risolvere in maniera efficiente interrogazioni formulate in un sottoinsieme di XPath su documenti XML con attributi e riferimenti.

[5]Il linguaggio XPath è un linguaggio per l'interrogazione di documenti XML che utilizza il concetto di albero per la formulazione delle interrogazioni. Questo linguaggio verrà presentato nel prossimo Capitolo.

Capitolo 2

Interrogazioni in XPath

2.1 XPath 1.0

XPath (XML Path language) è un linguaggio per l'interrogazione di documenti XML definito dal consorzio W3C in [10]. E' un linguaggio che consente di ricercare elementi all'interno di un documento XML rappresentato tramite un albero radicato etichettato. In XPath ogni elemento di un documento XML è considerato come un nodo dell'albero associato al documento. Prima di procedere oltre nella specifica di questo linguaggio introduciamo alcuni termini fondamentali:

- *memory tree*: è la rappresentazione, nella memoria del calcolatore, dell'albero associato al documento XML;
- *nodo*: qualsiasi elemento del *memory tree*. Possono esistere fondamentalmente tre tipologie di nodi: elemento, attributo e testo;
- *root*: il nodo radice del *memory tree* che contiene tutta la gerarchia del documento XML. E' un nodo speciale, indicato con il carattere /, che ha come unico figlio il primo nodo del documento XML.
- *document order*: ordine del documento, ciò significa che i nodi vengono inseriti nell'albero con lo stesso ordine del documento XML (in pratica l'albero viene costruito con un algoritmo di visita anticipata leggendo gli elementi del documento XML dall'alto verso il basso).

Il linguaggio XPath è molto utilizzato in ambito XML anche come linguaggio di base presente in altri tipi di tecnologie; ad esempio in:

- **XQuery**, un linguaggio per la ricerca e l'aggregazione, per future elaborazioni, di documenti XML (tipo operazioni di filtraggio, raggruppamento, ordinamento, formattazione, etc.).
- **XSLT** (eXtensible Stylesheet Language for Transformation), un linguaggio per la trasformazione di documenti XML in altri documenti XML o in altri formati.
- **XPointer**, per la realizzazione di collegamenti tra elementi di uno o più documenti XML.

In questo Capitolo analizzeremo soltanto una porzione del linguaggio XPath nella versione 1.0; per una visione completa di XPath si consiglia la lettura delle specifiche del consorzio W3C [10] e del libro di Holzner [11].

2.2 Percorsi XPath e Assi

Per poter comprendere il linguaggio XPath è necessario considerare un documento XML come una struttura gerarchica ad albero.

Ad esempio considerando il documento `bank.xml` riportato nel Paragrafo 1.4 si possono formulare le seguenti conclusioni:

- il primo nodo dell'albero, figlio del nodo *root*, è rappresentato dall'elemento `bank`;
- il primo ed il secondo nodo `customer` hanno come genitore il nodo `bank`;
- il nodo `account` è costituito da due nodi figli rappresentati dagli elementi `branch-name` e `balance`;
- il secondo nodo `customer` ha come fratello destro il primo nodo `account` e come fratello sinistro il primo nodo `customer`.

Un'interrogazione (*query*) in XPath è la specifica di un percorso nel *memory tree* di un documento XML finalizzato alla ricerca di uno o più elementi.

Un percorso XPath è costituito da un insieme di passi (*step*) separati dal carattere /. Ogni step è costituito da un asse (*axis*), da un separatore rappresentato dai caratteri :: (*axis separator*), da un *node test* e da uno o più filtri o predicati (*predicate*), così come riportato in Figura 2.1.

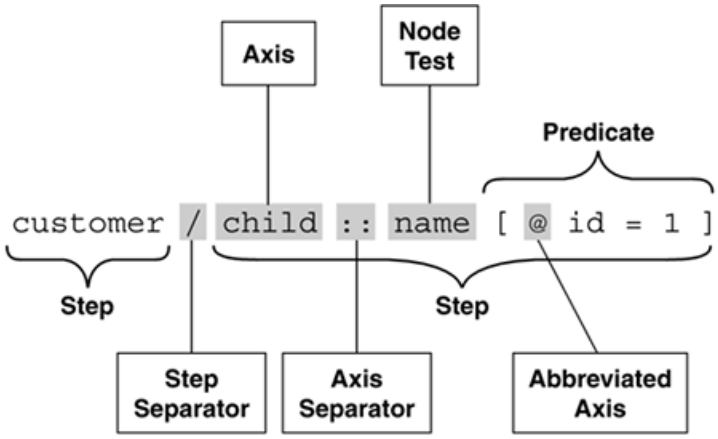

Figura 2.1: Anatomia di una query XPath

L'asse (*axis*) è il tipo di percorso, all'interno dell'albero XML, che deve essere elaborato per la ricerca del *node test*. Ad esempio l'asse `child` identifica il percorso costituito da tutti i nodi figli del nodo attuale. Il *node test* è il nome dell'elemento XML che si stà cercando, ad esempio in Figura 2.1 si stanno cercando elementi con *tag* pari a `name` (esiste un *node test* particolare indicato con il carattere $*$ che identifica qualsiasi tipo di elemento a prescindere dall'etichetta del suo nodo). Infine il filtro o predicato (*predicate*) indentifica quali sono gli elementi XML che devono essere presi in considerazione nel risultato finale.

Per poter formulare un'interrogazione XPath è necessario descrivere il percorso a partire dal nodo *root* rappresentato con il carattere /; successivamente si possono inserire una serie di *step* per la navigazione all'interno del documento XML. L'insieme dei nodi elaborati in ogni *step* è chiamato *context node*. All'inizio di un'interrogazione

XPath il *context node* è costituito solo dal nodo *root*.

Il risultato di un'interrogazione XPath è un *context node*, eventualmente vuoto, di nodi dell'albero XML.

Facciamo alcuni esempi considerando, come al solito, il documento `bank.xml` riportato nel Paragrafo 1.4:

- l'interrogazione `/child::bank/child::customer` restituisce tutti i nodi dell'albero XML etichettati con `customer` che discendono dal nodo *root* (`/`) e che hanno un padre etichettato con `bank`.
- l'interrogazione `/descendant::customer` restituisce tutti i nodi etichettati con `customer` discendenti del nodo *root* (/). Si noti che questa *query* è diversa da quella precedente poichè in quest'ultimo caso i nodi etichettati con `customer` vengono ricercati in tutto il documento XML e non solo al secondo livello dell'albero come figli dell'elemento `bank`.
- l'interrogazione `/descendant::customer[attribute::type='old']` restituisce tutti i nodi etichettati con `customer` che hanno un attributo etichettato con `type` il cui valore è pari a `old`. In questo caso la presenza del filtro serve per selezionare tra i risultati di `/descendant::customer` i nodi che soddisfano il predicato.
- l'interrogazione `/descendant::account/child::*` restituisce tutti i nodi figli dell'elemento `account`.
- l'interrogazione `/descendant::*` restituisce tutti i nodi del documento XML.

Gli assi fondamentali che possono essere utilizzati in un'interrogazione XPath 1.0 sono i seguenti:

- **self**: restituisce il *context node*;
- **parent**: restituisce i nodi genitori dei nodi presenti nel *context node*;
- **child**: restituisce i nodi figli dei nodi presenti nel *context node*;

- **descendant**: restituisce i nodi discendenti dei nodi presenti nel *context node* esclusi i nodi attributo;
- **descendant-or-self**: restituisce lo stesso risultato di **descendant** più il *context node* stesso;
- **ancestor**: restituisce i nodi antenati dei nodi presenti nel *context node* fino ad arrivare al nodo *root*;
- **ancestor-or-self**: restituisce lo stesso risultato di **ancestor** più il *context node* stesso;
- **attribute**: restituisce tutti i nodi attributo del *context node*;
- **following**: restituisce tutti i nodi che seguono, secondo l'ordine del documento, i nodi del *context node* (sono esclusi dal risultato i nodi attributo);
- **following-sibiling**: restituisce tutti i fratelli che seguono, secondo l'ordine del documento, i nodi del *context node* (sono esclusi dal risultato i nodi attributo);
- **preceding**: restituisce tutti i nodi che precedono, secondo l'ordine del documento, i nodi del *context node* (sono esclusi dal risultato i nodi attributo);
- **preceding-sibiling**: restituisce tutti i fratelli che precedono, secondo l'ordine del documento i nodi del *context node* (sono esclusi dal risultato i nodi attributo);

All'interno di un filtro possono essere utilizzati gli operatori riportati in Figura 2.2 per selezionare particolari elementi del *context node* che soddisfano il predicato (alcuni operatori devono essere specificati con le *entity-name* per rispettare la condizione *well formed* dei documenti XML).

Operatore	**Significato**
=	uguaglianza
!=	disuguaglianza
>	maggiore (>)
<	minore (<)
>=	maggiore o uguale (>=)
<=	minore o uguale (<=)
and	AND logico
or	OR logico
not()	NOT logico
position()	posizione ordinale del nodo
|	unione tra condizioni
+	addizione
-	sottrazione
*	moltiplicazione
div	divisione
mod	resto della divisione

Figura 2.2: Gli operatori utilizzabili all'interno dei filtri XPath

Finora abbiamo utilizzato il *node test* come etichetta di un elemento XML o con il carattere $*$ per indicare qualsiasi elemento del documento XML. In XPath esistono altri *node test* come ad esempio `text()` che consente di restituire il valore di un elemento (ossia la stringa associata a quell'elemento).

Ad esempio la seguente interrogazione `/descendant::surname/child::text()` sul documento `bank.xml` riportato nel Paragrafo 1.4 restituirà, come risultato, le stringhe "Turing" e "Newton".

2.3 Sintassi XPath abbreviata

Esistono alcuni assi XPath, come ad esempio `descendant`, che vengono utilizzati più spesso degli altri; per questo motivo lo standard XPath ha previsto una sintassi abbreviata per alcuni assi che consente di ridurre notevolmente le dimensioni di una query.

Di seguito sono riportate le abbreviazioni consentite nel linguaggio XPath:

- `child::a` è abbreviato in `a` (`child` è l'asse predefinito);
- `/descendant::a` è abbreviato con `//a`;
- `attribute::a` è abbreviato con `@a`;
- `self::*` è abbreviato con un punto (.);
- `parent::*` è abbreviato con due punti (..);

Da notare la similitudine delle abbreviazioni di `self::*` e `parent::*` con il punto (.) ed il doppio punto (..) presenti nella struttura delle directory di un sistema Unix.

Tramite questa sintassi abbreviata la dimensione delle query può essere ridotta notevolmente; ad esempio l'interrogazione seguente:

```
/descendant::a/child::b[attribute::c]/parent::*[self::* = 'w']
```

può essere ridotta nell'interrogazione:

```
//a/b[@c]/..[.  = 'w']
```

2.4 La funzione id()

La funzione `id()` è una funzione XPath fondamentale per le finalità di questa tesi di laurea perchè è quella che consente di "saltare" all'interno di un documento XML a seconda dei riferimenti specificati nello schema DTD. Attravero questa funzione è possibile selezionare qualsiasi elemento del documento XML che abbia un attributo di tipo ID con valore uguale al parametro specificato nella funzione. Per poter utilizzare la funzione `id()` è necessario utilizzare un documento XML con un schema DTD associato, altrimenti un *engine*[1] XPath non riuscirebbe ad interpretare correttamente la tipologia degli attributi ID, IDREF e IDREFS.

La funzione `id()` può essere utilizzata in due modi:

- in maniera *diretta* specificando un valore all'interno della funzione, ad esempio `id('A1')`;
- in maniera *indiretta* specificando un'interrogazione XPath che restituisca un attributo di tipo IDREF o IDREFS, ad esempio `id(/descendant::customer/attribute::accounts)`.

Nel primo caso la funzione `id('A1')` restituisce il nodo dell'albero XML con attributo di tipo ID e valore pari ad `'A1'`.

Nel secondo caso la funzione `id(/descendant::customer/attribute::accounts)` restituisce l'insieme dei nodi dell'albero XML con attributo di tipo ID pari al valore degli attributi `accounts` degli elementi `customer` del documento.

L'utilizzo di questa funzione XPath e di uno schema DTD associato ad un file XML ci consentono di lavorare sui documenti XML con attributi e riferimenti così come avevamo specificato nel Paragrafo 1.5.

Lo studio della complessità algoritmica derivante dall'utilizzo di questa funzione XPath verrà presentato nel prossimo Capitolo, dove dimostreremo che è possibile risolvere, con complessità lineare, alcune interrogazioni XPath su documenti XML con attributi e riferimenti.

[1] Con il termine *engine* intendiamo un software in grado di elaborare un'interrogazione.

Capitolo 3

Un algoritmo efficiente per un sottoinsieme di XPath

In questo capitolo sarà presentato un algoritmo efficiente per la risoluzione di un sottoinsieme di interrogazioni XPath che individueremo in seguito con il nome di SXPath (Standard XPath).

Per poter risolvere interrogazioni nel linguaggio SXPath utilizzeremo in realtà un'estensione di questo linguaggio che chiameremo EXPath (Extended XPath) ed una funzione di traduzione ϕ dal linguaggio SXPath verso EXPath.

Questo modo di operare appare interessante in quanto da un lato consente di sperimentare sul linguaggio XPath attraverso l'introduzione di nuove potenzialità in EXPath e dall'altro di rispettare gli standard XPath del consorzio W3C [10] tramite l'utilizzo del linguaggio SXPath.

Proprio in ragione di quanto appena sottolineato definiremo la sintassi dei linguaggi SXPath e EXPath, la semantica del linguaggio EXPath e la semantica del linguaggio SXPath attraverso quella di EXPath con l'utilizzo della funzione di traduzione $\phi : SXPath \mapsto EXPath$.

I documenti XML che saranno utilizzati contengono attributi e riferimenti di tipo ID, IDREF e IDREFS. In questo modo, come abbiamo già visto nel Paragrafo 1.5, si realizzano delle strutture dati simili a dei *grafi* tramite la creazione di relazioni tra elementi di un documento XML.

3.1 Notazione

In questo capitolo faremo uso della seguente notazione per indicare alcune relazioni matematiche.

Definizione 1 (Relazione inversa)**.** *Sia R una relazione, la sua relazione inversa è indicata con R^{-1} ed è definita nel modo seguente $R^{-1} = \{(m, n) \mid (n, m) \in R\}$.*

Definizione 2 (Chiusura transitiva)**.** *Sia R una relazione, la sua chisura transitiva è indicata con R^+ ed è definita nel modo seguente $R^+ = \bigcup_{i=1}^{\infty} R^i$.*

Definizione 3 (Chiusura riflessiva e transitiva)**.** *La chiusura riflessiva e transitiva di una relazione R su un insieme A è indicata con R^* ed è definita nel modo seguente $R^* = R^+ \cup I_A$, dove I_A è la relazione identità su A, $I_A = \{(x, x) \mid x \in A\}$.*

3.2 Sintassi SXPath e EXPath

SXPath è un sottoinsieme di XPath che estende il linguaggio Core XPath definito originariamente in [1]. Il linguaggio Core XPath è stato introdotto come linguaggio di sintesi delle principali caratteristiche di XPath. All'interno di questo linguaggio sono definiti i principali strumenti di navigazione (assi) di un documento XML con la presenza dei filtri o predicati (*[predicate]*), potenti strumenti in grado di effettuare ricerche nelle ricerche. Le funzioni aritmetiche, booleane, di gestione delle stringhe e dei node-set non sono presenti all'interno di Core XPath. SXPath è stato elaborato partendo da Core XPath con l'aggiunta degli attributi, della funzione id() e dell'operatore di uguaglianza (=) all'interno dei filtri. Di seguito è riportata la sua sintassi:

Definizione 4 (Sintassi SXPath)**.** *Sia Σ l'insieme delle etichette (tag) relative agli elementi e agli attributi di un documento XML. Un'interrogazione SXPath è una formula (query) generata dalla prima clausola della seguente definizione ricorsiva:*

$$
\begin{aligned}
query &= /path \\
path &= step(/step)^* \mid pointer(/step)^* \mid pointer[filter](/step)^* \\
pointer &= id('s') \mid id(path) \\
step &= axis::a \mid axis::a[filter] \\
filter &= path \mid filter = 's' \mid filter\ and\ filter \mid filter\ or\ filter \mid not\ filter \\
axis &= self \mid attribute \mid child \mid parent \mid descendant \mid descendant\text{-}or\text{-}self \mid \\
&\quad ancestor \mid ancestor\text{-}or\text{-}self \mid following \mid following\text{-}sibling \mid \\
&\quad preceding \mid preceding\text{-}sibling
\end{aligned}
$$

dove $a \in \Sigma \cup \{*\}$ *e* $s \in$ *String, l'insieme delle stringhe alfanumeriche.*

Il simbolo $*$ denota qualsiasi componente (elemento, attributo o testo) di un documento XML.

Definizione 5 (Sintassi EXPath)**.** *Sia* Σ *l'insieme delle etichette (tag) relative agli elementi e agli attributi di un documento XML. Un'interrogazione EXPath è una formula (query) generata dalla prima clausola della seguente definizione ricorsiva:*

$$
\begin{aligned}
query &= /path \\
path &= step\ (/step)^* \mid id('s')(/step)^* \mid id('s')[path](/step)^* \mid \\
&\quad id('s_1')='s_2'(/step)^* \mid id('s_1')[path]='s_2'(/step)^* \mid \\
&\quad path\ and\ path \mid path\ or\ path \mid not\ path \\
step &= axis::a \mid axis::a[path] \mid axis::a='s' \mid axis::a[path]='s' \\
axis &= self \mid child \mid parent \mid self\text{-}attribute \mid attribute \mid parent\text{-}attribute \mid \\
&\quad descendant \mid descendant\text{-}or\text{-}self \mid ancestor \mid ancestor\text{-}or\text{-}self \mid \\
&\quad following \mid following\text{-}sibling \mid preceding \mid preceding\text{-}sibling \mid \\
&\quad next \mid next\text{-}sibling \mid previous \mid previous\text{-}sibling \mid id \mid id^{-1}
\end{aligned}
$$

dove $a \in \Sigma \cup \{*\}$ *e* s, s_1, $s_2 \in$ *String, l'insieme delle stringhe alfanumeriche.*

Anche qui il simbolo $*$ denota qualsiasi componente (elemento, attributo o testo) di un documento XML. Questa definizione estende la sintassi XPath con l'aggiunta degli assi *id*, id^{-1}, *self-attribute*, *parent-attribute*, *next*, *next-sibling*, *previous* e *previous-sibling*, degli operatori logici *and*, *or* e *not* sui path e dell'operatore di uguaglianza (=) negli step. L'introduzione dei nuovi assi *self-attribute* e *parent-attribute* consente di definire un linguaggio di interrogazione più simmetrico rispetto

a quello definito dal consorzio W3C nel quale, ad esempio, gli attributi non possono essere considerati come figli di un nodo elemento anche se il genitore di un nodo attributo è il nodo elemento[1]. Nella nostra definizione abbiamo superato questa asimmetria, per noi gli assi *self* e *parent* vengono utilizzati solo sui nodi elemento e gli assi *self-attribute* e *parent-attribute* solo sui nodi attributo[2]. Per questo motivo un'interrogazione in standard XPath che utilizzi l'asse *parent* su di un attributo deve essere tradotta nel nostro linguaggio con l'asse *parent-attribute* (ad esempio l'interrogazione XPath /descendant::*/attribute::a/parent::* deve essere tradotta nel nostro linguaggio in /descendant::*/attribute::a/parent-attribute::*). Anche l'aggiunta dell'operatore di uguaglianza negli step si discosta leggermente dallo standard XPath, nel quale, ad esempio, l'interrogazione /descendant::a='s' non può essere utilizzata (l'istruzione XPath corrispondente è /descendant::a[self::*='s']). La sintassi che abbiamo presentato risulta quindi più potente tra le sintassi già proposte per XPath. Ad esempio l'istruzione /descendant::*/attribute::*='s' che consente di ottenere tutti gli attributi di un documento che hanno il valore uguale alla stringa 's' non può essere tradotta in XPath.

[1]Nelle specifiche XPath del W3C si legge quanto segue: "Ogni nodo elemento ha associato un insieme di nodi attributo; l'elemento è il genitore di ognuno di questi attributi però un nodo attributo non è figlio dell'elemento genitore".

[2]Così facendo il nostro linguaggio è più simile alle specifiche dello standard DOM (Document Object Model), definito sempre dal consorzio W3C in [12], rispetto allo standard XPath.

3.3 Il modello dei dati

Rappresentiamo un documento XML come un albero radicato etichettato con elementi (*tag*) , attributi e relativi valori di testo. Ogni nodo dell'albero rappresenta un *elemento*, un *attributo* o un *testo* del documento XML. Con il termine *testo* intendiamo le stringhe associate agli elementi o agli attributi di un documento XML. In particolare, gli attributi sono rappresentati come nodi figli di un nodo elemento ed i testi sono rappresentati come nodi figli di un nodo elemento o attributo. L'albero è costruito rispettando l'ordine degli elementi nel documento XML (*document order*), in particolare applicando l'algoritmo di visita anticipata sui nodi dell'albero si riottiene il documento XML originale. I nodi di tipo attributo, di un nodo elemento, vengono inseriti nell'albero prima dei nodi figli di tipo elemento, l'ordine di inserimento dei nodi attributo non è rilevante. I nodi di tipo testo vengono inseriti nell'albero nello stesso ordine in cui compaiono nel documento rispetto ai nodi figli (vedi esempio in Figura 3.1).

```
<person code='123'>
    text1
    <name age='42'>Alan</name>
    text2
    <surname>Turing</surname>
</person>
```

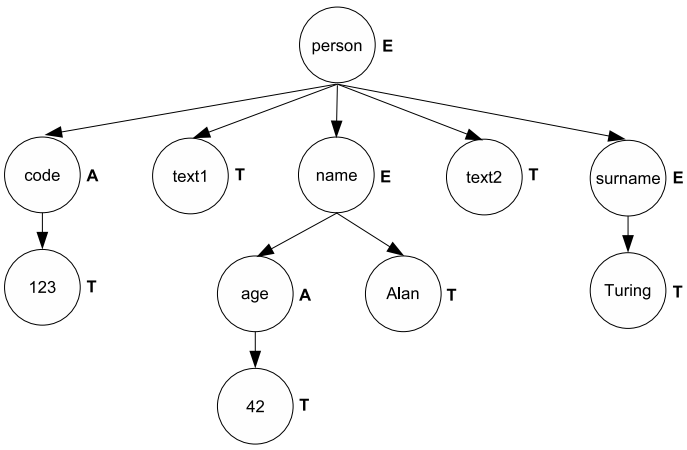

Figura 3.1: Esempio di documento XML con rappresentazione ad albero. Le lettere E, A e T indicano rispettivamente un nodo di tipo *elemento*, *attributo* e *testo*.

Indichiamo con Σ l'insieme delle etichette corrispondenti agli elementi e agli attributi di un documento XML. Nel nostro modello dei dati utilizziamo anche le seguenti funzioni: L che associa ad ogni componente elemento o attributo di un documento XML l'insieme dei nodi corrispondenti nell'albero, S che restituisce la stringa, il valore, associato ai nodi di tipo testo e *type* che indica la tipologia di ogni

nodo: elemento, testo, attributo generico o di tipo ID, IDREF o IDREFS. Tutti questi elementi costituiscono il nostro modello dei dati che indicheremo con il nome di *Albero XML*. Di seguito è riportata la sua definizione.

Definizione 6 (Albero XML)**.** *Sia Σ l'insieme delle etichette corrispondenti agli elementi e agli attributi di un documento XML. Un albero XML è un albero radicato etichettato T=(N, $R_{\downarrow}$, $R_{\rightarrow}$, L, S, type) dove N è l'insieme dei nodi dell'albero, $R_{\downarrow} \subseteq N \times N$ è l'insieme delle coppie di nodi (padre, figlio) dell'albero, $R_{\rightarrow} \subseteq N \times N$ è l'insieme delle coppie di nodi (n, fratello destro di n), $L : \Sigma \cup \{*\} \mapsto 2^N$ è la funzione che associa ad ogni componente elemento o attributo del documento XML i relativi nodi dell'albero, $S : N \mapsto String$ è la funzione che restituisce il valore della stringa (string-value[3]) associata al nodo e type: $N \mapsto \{$ element, text, attribute, id, idref, idrefs$\}$ è la funzione che associa ad ogni nodo la sua tipologia che può essere elemento (element), testo (text), attributo generico (attribute) o di tipo ID, IDREF e IDREFS secondo le informazioni riportate nel DTD (Document Type Definition) associato al documento XML in esame. String rappresenta l'insieme delle stringhe alfanumeriche.*

Secondo questa definizione risulta che $L(*) = N$ ossia è l'insieme di tutti i nodi dell'albero XML.

3.4 Semantica EXPath

Di seguito viene proposta una semantica di EXPath seguendo l'impostazione data in [1] e [13] con alcune varianti relative alle nuove estensioni del linguaggio.

Definizione 7 (Semantica EXPath)**.** *Sia T un albero XML (definito in 3.3). Sia root $\in$ N la radice dell'albero T. Definiamo la semantica di EXPath attraverso la funzione $f(T, q, C) \mapsto 2^N$ dove q è l'interrogazione (query) EXPath e $C \subseteq N$ è un sottoinsieme dei nodi dell'albero T nel quale applicare l'interrogazione q, nel modo seguente:*

[3]Per lo standard XPath lo *string-value* di un nodo è definito in modo ricorsivo dalla concatenazione delle stringhe dei suoi nodi figli di tipo *testo*, compreso il nodo stesso, e dalla concatenazione dello *string-value* di tutti i suoi nodi discendenti (ricordiamo che i nodi discendenti, per lo standard XPath, sono solo i nodi di tipo *elemento*). Questa definizione dello *string-value* per i nodi di tipo elemento si discosta dallo standard DOM [12] nel calcolo del metodo nodeValue.

$$
\begin{aligned}
f(T, axis :: a, C) &= \{n \in N \mid n \in [C]_{axis,T} \ \wedge \ n \in L(a)\} \\
f(T, axis :: a[path], C) &= \{n \in f(T, axis :: a, C) \mid \exists m \in N. \\
&\quad m \in f(T, path, \{n\})\} \\
f(T, axis :: a = 's', C) &= \{n \in f(T, axis :: a, C) \mid S(n) = 's'\} \\
f(T, axis :: a[path] = 's', C) &= \{n \in f(T, axis :: a[path], C) \mid S(n) = 's'\}
\end{aligned}
$$

$$
f(T, step_1/step_2, C) = f(T, step_2, f(T, step_1, C))
$$

$$
\begin{aligned}
f(T, id('s'), C) &= \{n \in N \mid \exists m \in N.\, type(m) = id \wedge \\
&\quad (n, m) \in R_{\downarrow} \ \wedge \ S(m) = 's'\} \\
f(T, id('s')[path], C) &= \{n \in f(T, id('s'), C) \mid \exists m \in N. \\
&\quad m \in f(T, path, \{n\})\} \\
f(T, id('s_1') = 's_2', C) &= \{n \in f(T, id('s_1'), C) \mid S(n) = 's_2'\} \\
f(T, id('s_1')[path] = 's_2', C) &= \{n \in f(T, id('s_1')[path], C) \mid S(n) = 's_2'\}
\end{aligned}
$$

$$
\begin{aligned}
f(T, path_1 \ and \ path_2, C) &= f(T, path_1, C) \cap f(T, path_2, C) \\
f(T, path_1 \ or \ path_2, C) &= f(T, path_1, C) \cup f(T, path_2, C) \\
f(T, not \ path, C) &= N \setminus f(T, path, C)
\end{aligned}
$$

$$
f(T, /path, C) = f(T, path, \{root\})
$$

dove $[C]_{axis,T} : C \times axis \times T \mapsto 2^N$ *è la funzione che elabora l'asse (axis) a partire dall'insieme* $C \subseteq N$*, definita nel modo seguente:*

$$
\begin{aligned}
[C]_{self,T} &= \{n \in C \mid type(n) = element\} \\
[C]_{child,T} &= \{n \in N \mid \exists c \in C.\, (c, n) \in R_{\downarrow} \ \wedge \ type(n) = element\} \\
[C]_{parent,T} &= \{n \in N \mid \exists c \in C.\, type(c) = element \ \wedge \ (c, n) \in R_{\uparrow} \wedge \\
&\quad type(n) = element\}
\end{aligned}
$$

$$
\begin{aligned}
[C]_{self-attribute,T} &= \{n \in C \mid type(n) \in \{\, attribute,\ id,\ idref,\ idrefs \,\}\} \\
[C]_{attribute,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in R_{\downarrow} \wedge \\
&\quad type(n) \in \{\, attribute,\ id,\ idref,\ idrefs \,\}\} \\
[C]_{parent-attribute,T} &= \{n \in N \mid \exists c \in C.\ type(c) \in \{\, attribute,\ id,\ idref,\ idrefs \,\} \\
&\quad \wedge\ (c,n) \in R_{\uparrow} \ \wedge\ type(n) = element\}
\end{aligned}
$$

$$
\begin{aligned}
[C]_{descendant,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in (R_{\downarrow})^{+} \wedge\ type(n) = element\} \\
[C]_{ancestor,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in (R_{\uparrow})^{+} \wedge\ type(n) = element\} \\
[C]_{descendant-or-self,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in (R_{\downarrow})^{*} \wedge\ type(n) = element\} \\
[C]_{ancestor-or-self,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in (R_{\uparrow})^{*} \wedge\ type(n) = element\} \\
[C]_{following-sibling,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in (R_{\rightarrow})^{+} \wedge\ type(n) = element\} \\
[C]_{preceding-sibling,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in (R_{\leftarrow})^{+} \wedge\ type(n) = element\} \\
[C]_{following,T} &= [\,[\,[\,C\,]_{ancestor-or-self,T}\,]_{following-sibling,T}\,]_{descendant-or-self,T} \\
[C]_{preceding,T} &= [\,[\,[\,C\,]_{ancestor-or-self,T}\,]_{preceding-sibling,T}\,]_{descendant-or-self,T} \\
[C]_{next,T} &= \{n \in N \mid \exists c \in C.\ pre(n) = min\,\{pre(m) \mid \\
&\quad m \in [\{c\}]_{following,T}\}\} \\
[C]_{previous,T} &= \{n \in N \mid \exists c \in C.\ pre(n) = max\,\{pre(m) \mid \\
&\quad m \in [\{c\}]_{preceding,T}\}\} \\
[C]_{next-sibling,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in R_{\rightarrow} \wedge\ type(n) = element\} \\
[C]_{previous-sibling,T} &= \{n \in N \mid \exists c \in C.\ (c,n) \in R_{\leftarrow} \wedge\ type(n) = element\}
\end{aligned}
$$

$$
\begin{aligned}
[C]_{id,T} &= \{n \in N \mid \exists c \in C.\ type(c) \in \{\, idref,\ idrefs \,\}\,. \\
&\quad \exists m \in N.\ type(m) = id\ \wedge (n,m) \in R_{\downarrow} \wedge \\
&\quad S(m) \in split(S(c))\} \\
[C]_{id^{-1},T} &= \{n \in N \mid \exists c \in C.\ type(c) = id.\ \exists m \in N. \\
&\quad type(m) \in \{\, idref,\ idrefs \,\}\ \wedge\ (n,m) \in R_{\downarrow} \wedge \\
&\quad S(c) \in split(S(m))\}
\end{aligned}
$$

dove $split('s') : String \mapsto 2^{String}$ *è la funzione che data una stringa s contenente degli spazi restituisce l'insieme delle sottostringhe di s prive di spazi (ad esempio* $split('C1\ C2\ A1') = \{'C1', 'C2', 'A1'\}$ *e nel caso in cui la stringa s sia priva di spazi* $split('s') = \{'s'\}$*) e* $pre(n) : N \mapsto \{1, \ldots, |N|\}$ *è la funzione che restituisce ad ogni nodo n la sua posizione nella visita anticipata dell'albero,* $R_{\leftarrow} = (R_{\rightarrow})^{-1}$*,*

$R_{\uparrow} = (R_{\downarrow})^{-1}$, R^{-1} è la relazione inversa di R, R^{+} è la chiusura transitiva di R e R^{*} è la chiusura riflessiva e transitiva di R.

Da questa definizione segue che ogni asse x ha un'inverso, ossia un'asse x^{-1} che pone in relazione il nodo n' elaborato dall'asse x con il nodo n che lo ha generato. In particolare può essere dimostrato il seguente Lemma (definito originariamente in [1]):

Lemma 1 (Relazione tra assi e loro inversi)**.** *Sia χ un asse EXPath. Per ogni coppia di nodi n,n' $\in N$, n χ n' se e solo se n' χ^{-1} n.*

Ad esempio *n child* n' se e solo se n' *parent* n, ossia n è figlio di n' se e solo se n' è genitore di n. In particolare si hanno le seguenti relazioni:

$$
\begin{aligned}
\text{self}^{-1} &= \text{self} \\
\text{child}^{-1} &= \text{parent} \\
\text{self-attribute}^{-1} &= \text{self-attribute} \\
\text{attribute}^{-1} &= \text{parent-attribute} \\
\text{descendant}^{-1} &= \text{ancestor} \\
\text{descendant-or-self}^{-1} &= \text{ancestor-or-self} \\
\text{following-sibling}^{-1} &= \text{preceding-sibling} \\
\text{following}^{-1} &= \text{preceding} \\
\text{next}^{-1} &= \text{previous} \\
\text{next-sibling}^{-1} &= \text{previous-sibling} \\
(\text{id}^{-1})^{-1} &= \text{id}
\end{aligned}
$$

3.5 Semantica SXPath

Per poter definire la semantica del linguaggio SXPath definiamo una funzione di traduzione $\phi : SXPath \mapsto EXPath$ ed utilizziamo la semantica del linguaggio EXPath.

Definizione 8 (Funzione di traduzione da SXPath a EXPath)**.** *Sia q un'interrogazione in SXPath, $q = /q_1/\ldots/q_n$ dove $q_i \in$ step ed n indica un numero naturale. La funzione di traduzione $\phi(q) : SXPath \mapsto EXPath$ è definita sui singoli step q_i di q nel modo seguente:*

$$\phi(\ /q_1/\ldots/q_n\ ) = /\phi(q_1)/\ldots/\phi(q_n)$$

dove

$$\phi(q_i) = \begin{cases} \phi(path)/id :: * & se \quad q_i = id(path) \\ parent\text{-}attribute :: a & se \quad q_{i-1}/q_i = attribute::b/parent::a \\ \phi(axis::a)[\phi(filter)] & se \quad q_i = axis::a[filter] \text{ e } (\text{ } axis \notin \{ attribute, self\text{-}attribute \} \text{ o } filter \neq parent::b(/p)\ ) \\ \phi(axis::a)[parent\text{-}attribute::b(/\phi(p))] & se \quad q_i = axis::a[filter] \text{ e } axis \in \{ attribute, self\text{-}attribute \} \text{ e } filter = parent::b(/p) \\ \phi(filter) = \text{'s'} & se \quad q_i = filter = \text{'s'} \\ \phi(filter_1) \ and \ \phi(filter_2) & se \quad q_i = filter_1 \ and \ filter_2 \\ \phi(filter_1) \ or \ \phi(filter_2) & se \quad q_i = filter_1 \ or \ filter_2 \\ not(\phi(filter)) & se \quad q_i = not(filter) \\ q_i & altrimenti \end{cases}$$

con a,b $\in \Sigma \cup \{*\}$ *e* $i = 1, \ldots, n$.

Definizione 9 (Semantica SXPath)**.** *Sia T un albero XML (definito in 3.3). Sia q un'interrogazione in SXPath e $\phi(q) : SXPath \mapsto EXPath$ la funzione di traduzione da SXPath a EXPath. Definiamo la semantica di SXPath attraverso la funzione $f(T, \phi(q), C) \mapsto 2^N$ definita, in precedenza, nella semantica di EXPath, dove $C \subseteq N$ è un sottoinsieme dei nodi dell'albero T nel quale applicare l'interrogazione $\phi(q)$.*

3.6 L'algoritmo

In questo paragrafo presentiamo un algoritmo efficiente per la risoluzione di interrogazioni EXPath e SXPath. L'algoritmo che sarà presentato è stato elaborato partendo dai lavori di Georg Gottlob, Christopher Koch e Pichler Pichler in [1, 14] per quel che riguarda l'idea generale di risoluzione di interrogazioni Core XPath in tempo lineare, per la tecnica di risoluzione dei filtri e della funzione id e id^{-1}, dai lavori di Jan HIdders e Philippe Michiels in [15] per l'implementazione di alcuni assi e dal lavoro di Torsten Grust ed altri in [16, 17, 18] per l'idea dell'utilizzo dei valori *pre* e *post* di ogni nodo dell'albero per velocizzare l'elaborazione delle interrogazioni XPath.

Per l'elaborazione delle interrogazioni XPath assumiamo di lavorare con un insieme di nodi ordinato, *document order*, e senza ripetizioni (tale insieme verrà indicato con la lettera C).

Per la memorizzazione degli insiemi di nodi dell'albero XML utilizziamo una struttura dati a lista tramite le seguenti funzioni elementari:

- NewList(); inizializza una nuova lista
- DelFirst(C); restituisce ed elimina il primo elemento della lista C
- DelLast(C); restituisce ed elimina l'ultimo elemento della lista C
- AddAfter(C,n); aggiunge l'elemento n alla fine della lista C
- AddListAfter(C,L); aggiunge la lista L alla fine della lista C
- AddBefore(C,n); aggiunge l'elemento n all'inizio della lista C
- AddListBefore(C,L); aggiunge la lista L all'inizio della lista C
- First(C); restituisce il primo elemento della lista C
- Last(C); restituisce l'ultimo elemento della lista C

Per l'elaborazione degli assi XPath ipotizziamo di avere a disposizione l'albero del documento XML con le seguenti funzioni elementari:

- first-child(n,type); restituisce il primo figlio (a partire da sinistra) del nodo n di tipo *type*
- right-sibling(n,type); restituisce il fratello destro del nodo n di tipo *type*
- left-sibling(n,type); restituisce il fratello sinistro del nodo n di tipo *type*
- parent-node(n,type); restituisce il nodo padre del nodo n di tipo *type*

dove $type \in \{all, element, attribute\}$. Il parametro type consente quindi di ottenere nodi di tipo elemento (*element*), attributo (*attribute*) o di qualsiasi tipo (*all*).

Assumiamo inoltre che tutte le operazioni elementari appena presentate sulla lista dei nodi da elaborare (C) e sull'albero XML possano essere elaborate con complessità computazionale costante *O(1)*.

Con queste ipotesi dimostreremo che l'elaborazione delle interrogazioni EXPath e SXPath può essere ottenuta con complessità computazionale lineare nella dimensione dell'interrogazione e nella dimensione dell'albero XML.

In maniera formale se indichiamo con q l'interrogazione EXPath o SXPath e con T l'albero XML la complessità computazionale dell'elaborazione di q sarà al più $O(k \cdot n)$ dove k è la lunghezza di q ed n è la cardinalità di T.

3.6.1 Calcolo del valore *pre/post* per i nodi dell'albero XML

Per poter risolvere in maniera efficiente l'elaborazione di una query in XPath sono state presentate, in questi ultimi anni, diverse tecniche basate principalmente sull'utilizzo dei valori di *pre/post*. Questi valori di *pre/post* sono determinati dall'ordine della visita *anticipata* (*pre*) e *posticipata* (*post*) dei nodi di una struttura dati ad albero.

In particolare nel caso di documenti XML, il nostro modello dei dati prevede che per ogni nodo dell'albero vengono determinati e memorizzati i valori di *pre/post* con una coppia ordinata di numeri (*pre*, *post*) come nell'esempio di Figura 3.2, dove è riportato l'albero del documento XML presentato in Figura 3.1 a pag. 29.

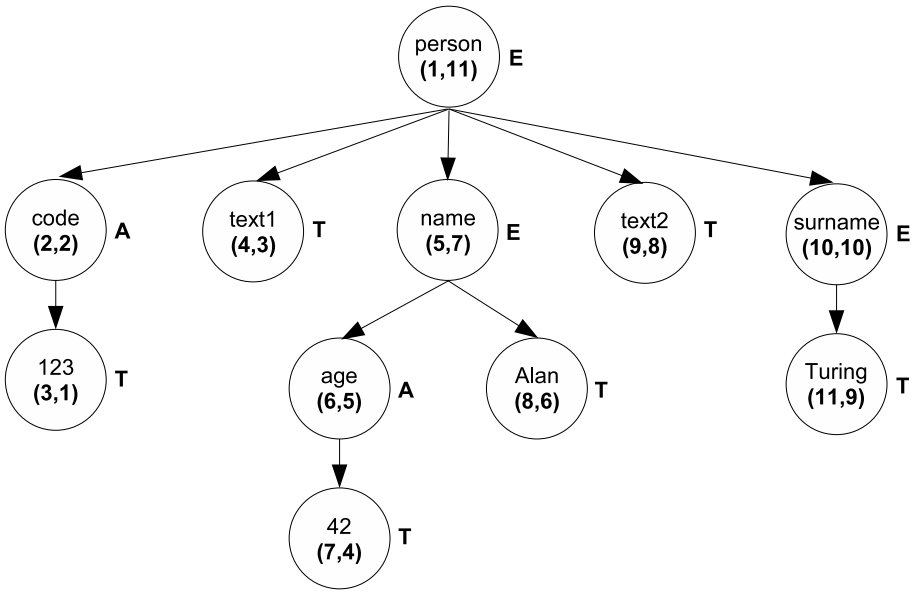

Figura 3.2: Esempio di albero XML con valori (*pre*, *post*) calcolati per ogni nodo.

Utilizzando queste informazioni su ogni nodo dell'albero molti ricercatori, tra i quali Torsten Grust, Maurice van Keulen, Jens Teubner, Jan Hidders, Philippe Michiels ed altri in [16, 18, 17, 15] sono riusciti a migliorare, in termini di complessità computazionale, gli algoritmi di risoluzione degli assi presenti all'interno di XPath. In particolare una volta determinati i valori (*pre*, *post*) per ogni nodo dell'albero è possibile utilizzare questa coppia di numeri per rappresentare i nodi su

di un piano cartesiano dove sull'asse delle ascisse vengono riportati i valori di *pre* e sull'asse delle ordinate i valori di *post* (vedi esempio in Figura 3.3).

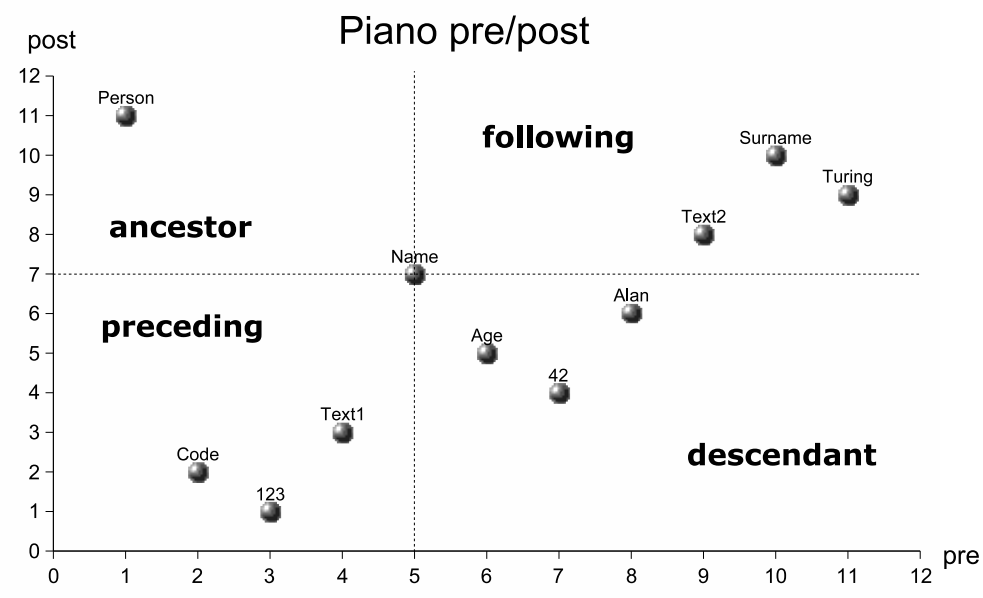

Figura 3.3: Esempio di piano *pre/post* dell'albero di Figura 3.2.

Osservando la posizione dei nodi all'interno di questo piano *pre/post* si possono trarre numerose informazioni utili per velocizzare l'elaborazione di alcuni assi XPath.

Ad esempio considerando il nodo **Name**, in Figura 3.3, con valori di (*pre*,*post*) pari a (5,7) si possono determinare i suoi nodi discendenti (*descendant*), antenati (*ancestor*), successori (*following*) e precedenti (*preceding*) suddividendo semplicemente il piano in quattro regioni tramite l'intersezione di due linee, una orizzontale e l'altra verticale, passanti per il nodo stesso.

In particolare per ogni nodo n si possono elaborare gli assi (*axes*) *descendant*, *ancestor*, *following* e *preceding* tramite un semplice confronto dei valori *pre/post* sui restanti nodi x dell'albero utilizzando la regola seguente:

- se $pre(x) > pre(n)$ e $post(x) > post(n)$ allora x è *following* di n
- se $pre(x) > pre(n)$ e $post(x) < post(n)$ allora x è *descendant* di n

- se $pre(x) < pre(n)$ e $post(x) < post(n)$ allora x è *preceding* di n
- se $pre(x) < pre(n)$ e $post(x) > post(n)$ allora x è *ancestor* di n

Nell'algoritmo presentato in questo Pragrafo vengono utilizzate queste tecniche di *pre/post* per ridurre, potare (*pruning*) i nodi da elaborare in un insieme *document order* di nodi.

3.6.2 La tecnica del *flag* numerico per la marcatura dei nodi

Per poter migliorare la velocità di elaborazione delle interrogazioni XPath oltre all'utilizzo della tecnica *pre/post* presentata in precedenza utilizziamo una nuova tecnica originale. Per ogni nodo dell'albero introduciamo un nuovo valore numerico, denominato *flag*, che verrà utilizzato all'interno dell'elaborazione dell'interrogazione XPath per marcare il passaggio in un nodo.

La tecnica dell'utilizzo di un *flag*, però di tipo booleano (vero/falso), per la marcatura dei nodi era già stata presa in considerazione da altri autori ma è stata subito scartata poichè ad ogni *step* di un'interrogazione XPath era presente il problema dell'azzeramento di questo valore per tutti i nodi dell'albero con conseguente esplosione della complessità computazionale in termini esponenziali $O(n^k)$ con n dimensione dell'albero XML e k dimensione dell'interrogazione XPath.

L'idea è quella di utilizzare un *flag* con valore di tipo numerico al posto di un valore di tipo booleano.

Durante l'interrogazione XPath utilizziamo un contatore (denominato con $CONT$) che verrà incrementato ad ogni *step* dell'interrogazione ed il cui valore verrà utilizzato per marcare il *flag* del nodo appena elaborato.

Questa tecnica consente di evitare di dover riazzerare ad ogni *step* i valori del *flag* per tutti i nodi dell'albero XML con conseguente risparmio di tempo.

Inizialmente il valore del *flag* per tutti i nodi è posto uguale a zero e, come abbiamo visto, per ogni step dell'interrogazione si utilizza un valore numerico $CONT$ incrementale. In questo modo per l'elaborazione di un singolo *step*, dato un nodo n dell'albero XML, è sufficiente verificare che *flag(n)* sia minore (oppure diverso) da $CONT$ per poter stabilire che il nodo n non sia stato ancora elaborato.

Per l'elaborazione di due o più interrogazioni XPath non è comunque necessario riazzerare, per ogni interrogazione, il valore del *flag* per tutti i nodi dell'albero XML, questa operazione può essere eseguita anche solo al momento in cui *CONT* ha raggiunto il valore massimo consentito (quest'aspetto verrà ripreso quando si introdurrà l'algoritmo *Evaluate* per l'elaborazione degli assi EXPath e durante la sezione di implementazione del Capitolo 4).

Questa tecnica del *flag* di tipo numerico risulta essere interessante poichè consente di costruire algoritmi molto semplici dal punto di vista computazionale anche su contesti differenti da quello esposto in questa tesi di laurea. Si pensi, ad esempio, alla possibilità di utilizzo di questa tecnica su strutture dati di tipo grafo per poter stabilire il passaggio in un nodo in *step* differenti.

3.6.3 Elaborazione dello *string-value* di un nodo dell'albero XML

Secondo le specifiche [10] del consorzio W3C lo *string-value* di un nodo elemento n è la concatenazione delle stringhe associate ai nodi di tipo testo discendenti del nodo n (questa definizione è stata già introdotta nel Paragrafo 3.3).

Nel caso di nodi attributo lo string-value è rappresentato dalla normalizzazione, secondo le raccomandazioni XML [3] del consorzio W3C, della stringa associata al nodo[4].

Per poter effettuare l'elaborazione dello string-value dei discendenti di un nodo n di tipo elemento introduciamo una procedura *AllText(n,value)* che restituisce nella variabile *value* la concatenazione di tutte le stringhe associate ai nodi di tipo testo discendenti del nodo n.

AllText (n,value)

```
1   n' ← first-child(n,all);
2   while n' ≠ NULL do
3       if type(n')=text then
4           value ← concat(value,key[n']);
5       else
6           if type(n')=element then
7               AllText(n',value);
8           endif
9       endif
10      n' ← right-sibling (n',all);
11  endw
```

In questa procedura abbiamo utilizzato le funzioni *concat()*, *first-child()* e *right-sibling()*. La funzione *concat(a,b)* serve semplicemente per concatenare le due stringhe a e b; le funzione *first-child()* e *right-sibling()* con il parametro *all* restituiscono rispettivamente il primo figlio e il fratello destro del nodo n senza nessuna discriminazione sulla tipologia del nodo ottenuto.

[4]Per semplicità e dal momento che non risulta strumentale agli obiettivi della tesi stessa, noi non effettueremo questa procedura di normalizzazione. Pertanto lo *string-value* di un nodo attributo per noi sarà semplicemente la stringa associata al figlio di tipo testo (*text*) del nodo attributo.

Il parametro *all* risulta indispensabile per poter ottenere lo *string-value* dei nodi discendenti rispettando l'ordine del documento.

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè al massimo effettuerà una visita completa dell'albero.

A questo punto possiamo introdurre la funzione **S(n)** che restituisce lo *string-value* di un nodo n qualsiasi.

S (n)

```
1   value ← ' ';
2   if type(n) = text then
3       value ← key[n];
4   else if type(n) = element then
5       AllText(n,value);
6   else
7       n' ← first-child(n,text);
8       value ← key[n'];
9   endif
10  return value;
```

La complessità computazionale della funzione $S(n)$ è pari, nel caso peggiore, a $O(n)$ dove n rappresenta la cardinalità dell'albero XML ossia il numero di nodi dell'albero.

Introduciamo ora una nuova funzione **Equal(C,s)** che utilizzando la funzione appena introdotta $S(n)$ consente di effettuare il confronto tra una lista C di nodi, *document order*, ed una stringa contenuta in s restituendo i nodi $n \in C$ che hanno lo *string-value* uguale ad s.

Equal (C,s)

```
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       if S(n) = s then
5           AddAfter(L,n);
6       endif
7   endw
8   return L;
```

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n^2)$ poichè per ogni nodo $n \in C$ è necessario effettuare l'elaborazione dello *string-value* $S(n)$ con complessità computazionale $O(n)$.

Questo risultato può essere migliorato di un fattore $O(n)$ tramite una pre-elaborazione dello *string-value* di ogni nodo in fase di costruzione dell'albero XML. Assumendo dunque che per ogni nodo sia già stato calcolato il suo *string-value* l'elaborazione della funzione **Equal(C,s)** risulterà avere una complessità computazionale $O(n)$ al posto di $O(n^2)$.

Lo svantaggio di questa pre-elaborazione è rappresentato dall'esigenza di dover memorizzare più stringhe ridondanti nell'albero XML con conseguente 'spreco' di memoria; infatti se consideriamo due nodi di tipo elemento n,n' con n' figlio di n allora lo *string-value* di n conterrà sicuramente lo *string-value* di n'.

In questa tesi si è preferito optare per il calcolo a *run-time* dello *string-value* dei nodi considerando che mediamente la presenza delle uguaglianze con stringhe all'interno di una query XPath risulta essere, per lo più, con nodi il cui *string-value* è rappresentato semplicemente dal valore di un nodo figlio di tipo testo (*text*), quindi in termini di complessità computazionale media il fattore $O(n^2)$ tende a ridursi a $O(n)$.

3.6.4 Elaborazione degli assi

In questa sezione presentiamo tutti gli algoritmi per l'elaborazione degli assi. Introduciamo una serie di funzioni che verranno utilizzate negli algoritmi sottostanti. Una prima funzione che risulterà particolarmente utile per elaborare interrogazioni del tipo *axis::a* è *tag(n)* che per ogni $n \in N$ restituisce il tag dell'elemento nel caso in cui n sia un nodo elemento ed il nome dell'attributo nel caso in cui n sia un nodo attributo.

Elaborazione dell'asse self::a

```
self (C,a)
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       if (type(n)=element) and ((tag(n)=a) or (a='*')) then
5           AddAfter(L,n);
6       endif
7   endw
8   return L;
```

Se indichiamo con m la cardinalità della lista C allora l'elaborazione dell'asse **self::a** ha una complessità computazionale $O(m)$ ossia è lineare nella dimensione della lista. Nel caso peggiore la complessità computazionale risulta essere *O(n)* dove n è la cardinalità dell'albero XML ossia il numero di nodi dell'albero.

Elaborazione dell'asse child::a

Per l'elaborazione dell'asse **child::a** utilizziamo una funzione *AllChildren(n,a)* che restituisce una lista *document order* dei figli del nodo n che hanno etichetta uguale ad a.

AllChildren (n,a)

```
1   L ← NewList();
2   n' ← first-child(n,element);
3   while n' ≠ NULL do
4       if (tag(n')=a) or (a='*') then
5           AddAfter(L,n');
6       endif
7       n' ← right-sibling(n',element);
8   endw
9   return L;
```

La complessità computazionale di questa funzione, nel caso pessimo, è $O(n)$ dove n è la cardinalità dell'albero XML.

Introduciamo ora la funzione vera e propria per l'elaborazione dell'asse **child::a**.

child (C,a)

```
1   L ← NewList();
2   S ← NewList();
3   while not(empty(C)) do
4       n ← First(C);
5       if empty(S) then
6           AddListBefore(S,AllChildren(n,a));
7           n ← DelFirst(C);
8       else if pre(First(S)) <= pre(n) then
9           AddAfter(L,DelFirst(S));
10      else
11          AddListBefore(S,AllChildren(n,a));
12          n ← DelFirst(C);
13      endif
14  endw
15  if not(empty(S)) then
16      AddListAfter(L,S);
17  endif
18  return L;
```

Per poter recuperare i figli dei nodi contenuti nella lista C rispettando l'ordine del documento XML è necessario analizzare per ogni coppia di nodi n, $n' \in C$ con $n < n'$ se i figli di n precedano o seguano i figli di n'. Per poter stabilire l'ordine dei nodi utilizziamo la funzione *pre(n)* (riga 8 dell'algoritmo) che restituisce la posizione del nodo n secondo l'algoritmo di visita anticipata sull'albero XML (questa posizione

pre(n) rappresenta proprio l'ordine del documento, *document order*). I figli del nodo attuale vengono memorizzati in una lista temporanea, denominata con la lettera S, che simula una struttura dati di tipo stack. Questa lista viene utilizzata per inserire, in maniera ordinata, i nodi nella lista L a seconda che essi precedino o seguino i figli dei nodi successori della lista C. La parte principale dell'algoritmo è composta da un ciclo while (righe 3-14) nel quale vengono determinati i figli dei nodi contenuti nella lista C in input tramite la funzione *AllChildren (n,a)* nelle righe 6 e 11. L'ultima parte dell'algoritmo consente di inserire alla fine della lista L (che rappresenta l'output della funzione) la lista temporanea S se non risulta vuota (righe 15-17). Se indichiamo con n la cardinalità dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè non esistono due elementi che hanno figli in comune.

Elaborazione dell'asse parent::a

```
parent (C,a)
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       n' ← parent-node(n,element);
5       if (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
6           AddAfter(L,n');
7           flag(n') ← CONT;
8       endif
9   endw
10  return L;
```

In questo algoritmo vengono determinati i nodi padre/genitore (*parent*) dei nodi elemento contenuti nella lista C. Per eliminare il problema derivante da una duplicazione dei nodi genitori nel risultato, ad esempio quando due nodi $n,n' \in C$ hanno il *parent* in comune, utilizziamo una tecnica di marcatura dei nodi tramite l'utilizzo di un *flag*. La funzione *flag(n)* restituisce il valore numerico del *flag* per il nodo n; ad ogni elaborazione di uno step dell'interrogazione EXPath il valore del *flag* corrente viene incrementato e memorizzato nella variabile globale $CONT$. Questa tecnica consente di evitare di dover riazzerare il valore del *flag* per ogni nodo dell'albero ad ogni step dell'interrogazione. L'operazione di riazzeramento del valore *flag* per ogni nodo

dell'albero XML dovrà essere effettuata solo quando si arriva al valore massimo e dimensionando opportunamente tale valore in modo che il massimo sia sempre minore della lunghezza dell'interrogazione (in termini di step) otteniamo un'elaborazione efficiente. Se indichiamo con n la cardinalità dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ grazie all'utilizzo del *flag* che consente di non elaborare due volte gli stessi nodi.

Elaborazione dell'asse self-attribute::a

self-attribute (C,a)

```
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       if (type(n) ∈ {attribute, id, idref, idrefs}) and ((tag(n)=a) or (a='*')) then
5           AddAfter(L,n);
6       endif
7   endw
8   return L;
```

Se indichiamo con n la cardinalità dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$.

Elaborazione dell'asse attribute::a

attribute (C,a)

```
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       n' ← first-child(n,attribute);
5       while n' ≠ NULL do
6           if (tag(n')=a) or (a='*') then
7               AddAfter(L,n');
8           endif
9           n' ← right-sibling(n',attribute);
10      endw
11  endw
12  return L;
```

L'elaborazione dell'asse **attribute::a** viene realizzata determinando per ogni no-

do $n \in C$ i suoi attributi ossia i nodi figli di tipo attributo (linee 4-10 dell'algoritmo). Le funzioni *first-child* e *right-sibling* consentono di specificare tramite il secondo parametro la tipologia del nodo figlio e fratello da elaborare, in questo caso di tipo attributo (*attribute*). Se indichiamo con n la cardinalità dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a *O(n)* poichè non esistono elementi con attributi in comune.

Elaborazione dell'asse parent-attribute::a

```
parent-attribute (C,a)
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       if type(n) ∈ {attribute, id, idref, idrefs} then
5           n' ← parent-node(n,element);
6           if (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
7               AddAfter(L,n');
8               flag(n') ← CONT;
9           endif
10      endif
11  endw
12  return L;
```

L'elaborazione dell'asse **parent-attribute::a** viene realizzata determinando per ogni nodo $n \in C$ di tipo attributo il relativo nodo padre di tipo elemento. Anche in questo caso, come per l'algoritmo **parent(C,a)**, utilizziamo la tecnica del *flag* tramite la variabile globale di tipo $CONT$ poichè possono esistere più attributi con lo stesso padre/genitore. Se indichiamo con n la cardinalità dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a *O(n)* grazie all'utilizzo del *flag* che consente di non elaborare due volte gli stessi nodi.

Elaborazione degli assi descendant::a e descendant-or-self::a

Per l'elaborazione degli assi **descendant::a** e **descendant-or-self::a** utilizziamo una procedura ricorsiva denominata *AddDescendant(L,n,a)* che per un dato nodo n restituisce la lista L, *document order*, dei discendenti. Anche per questa funzione abbiamo utilizzato la tecnica del *flag* per la marcatura dei nodi già elaborati. In questo modo si evita di rielaborare più volte i nodi che risultano comuni discendenti di due o più nodi differenti.

AddDescendant (L,n,a)

```
1   n' ← first-child(n,element);
2   while (n' ≠ NULL) and (flag(n') < CONT) do
3       if (tag(n')=a) or (a='*') then
4           AddAfter(L,n');
5       endif
6       flag(n') ← CONT;
7       AddDescendant(L,n',a);
8       n' ← right-sibling(n',element);
9   endw
```

Di seguito è riportato il vero e proprio algoritmo per l'elaborazione degli assi **descendant::a** e **descendant-or-self::a**. Quest'unico algoritmo è in grado di implementare tutte e due gli assi *descendant* e *descendant-or-self* grazie all'utilizzo del parametro *self*. Se *self* è impostato a *true* (vero) la funzione elabora l'asse *descendant-or-self*, altrimenti viene elaborato l'asse *descendant*.

descendant (C,a,self)

```
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       if (self) and ((tag(n)=a) or (a='*')) then
5           AddAfter(L,n);
6           flag(n) ← CONT;
7       endif
8       AddDescendant (L,n,a);
9   endw
10  return L;
```

Se indichiamo con n la cardinalità dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè ogni nodo è visitato al

più una sola volta (grazie all'utilizzo del *flag*).

Elaborazione degli assi ancestor::a e ancestor-or-self::a

ancestor (C,a,self)

```
1   L ← NewList();
2   while not(empty(C)) do
3       n ← DelFirst(C);
4       S ← NewList();
5       if self then
6           n' ← n;
7       else
8           n' ← parent-node(n,element);
9       endif
10      while (n' ≠ NULL) and (flag(n') < CONT) do
11          if (tag(n')=a) or (a='*') then
12              AddBefore(S,n');
13          endif
14          flag(n') ← CONT;
15          n' ← parent-node(n,element);
16      endw
17      AddListAfter(L,S);
18  endw
19  return L;
```

L'algoritmo è composto da 2 cicli *while* annidati, quello più esterno (linee 2-18) effettua la scansione di tutti gli elementi $c \in C$, mentre il ciclo interno (linee 10-16) effettua l'elaborazione dei nodi ascendenti del nodo c. All'interno di quest'ultimo *while* i nodi discendenti vengono memorizzati in una lista temporanea S in ordine inverso, *LIFO* (Last In First Out), simulando una struttura dati di tipo *stack*. Una volta elaborati tutti i nodi ascendenti del nodo attuale c essi vengono inseriti in fondo alla lista L, tramite la funzione *AddListAfter(L,S)* (linea 17), che rappresenta l'output della funzione.

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè ogni nodo è visitato al più una sola volta (grazie all'utilizzo del *flag*).

Elaborazione dell'asse following::a

Per poter elaborare l'asse **following::a** introduciamo una funzione ricorsiva *AddAllDescendant(L,n,a)* che restituisce i discendenti del nodo n con *tag(n)=a* senza utilizzare il confronto con il valore del *flag* così come avveniva per la funzione *AddDescendant(L,n,a)*.

AddAllDescendant (L,n,a)

```
1   n' ← first-child(n,element);
2   while n' ≠ NULL do
3       if (tag(n')=a) or (a='*') then
4           AddAfter(L,n');
5       endif
6       AddAllDescendant(L,n',a);
7       n' ← right-sibling(n',element);
8   endw
```

Di seguito è riportato l'algoritmo per l'elaborazione dell'asse **following::a**.

following (C,a)

```
1   L ← NewList();
2   if not(empty(C)) then
3       n ← DelFirst(C);
4       while (not(empty(C)) and post(First(C)) < post(n)) do
5           n ← DelFirst(C);
6       endw
7       while n ≠ NULL do
8           n' ← right-sibling(n,element);
9           while n' ≠ NULL do
10              if (tag(n')=a) or (a='*') then
11                  AddAfter(L,n');
12              endif
13              AddAllDescendant (L,n',a);
14              n' ← right-sibling(n',element);
15          endw
16          n ← parent-node(n,element);
17      endw
18  endif
19  return L;
```

Per l'elaborazione dell'asse **following::a** su una lista ordinata (*document order*) di nodi C è sufficiente effettuare il calcolo sul primo elemento n che ha il più basso

valore locale di *post(n)*, ossia sull'elemento rappresentato dal nodo che si trova più a sinistra nell'albero XML rispetto agli altri nodi della lista C (righe 4-6 dell'algoritmo).

Per l'elaborazione di quest'asse non abbiamo utilizzato la tecnica del *flag* poichè il calcolo del *following* su di un singolo nodo n non produce ripetizione di nodi. Questo è anche il motivo che ci ha spinto ad utilizzare la funzione *AddAllDescendant* al posto di *AddDescendant*; infatti senza l'utilizzo del *flag* l'algoritmo *AddAllDescendant* risulta essere più veloce rispetto a *AddDescendant*.

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a *O(n)* poichè ogni nodo è visitato al più una sola volta.

Elaborazione dell'asse following-sibling::a

```
following-sibling (C,a)
1   L ← NewList();
2   H ← NewList();
3   while not(empty(C)) do
4       S ← NewList();
5       n ← DelFirst(C);
6       n' ← right-sibling(n,element);
7       while (n' ≠ NULL) and (flag(n') < CONT) do
8           if (tag(n')=a) or (a='*') then
9               if (not(empty(C)) and post(First(C)) < post(n') then
10                  AddAfter(S,n');
11              else
12                  while not(empty(H)) and pre(First(H)) < pre(n') do
13                      AddAfter(L,DelFirst(H));
14                  endw
15                  AddAfter(L,n');
16              endif
17          endif
18          flag(n') ← CONT;
19          n' ← right-sibling(n',element);
20      endw
21      AddListBefore (H,S);
22  endw
23  AddListAfter(L,H);
24  return L;
```

L'algoritmo per l'elaborazione dell'asse **following-sibling::a** utilizza due liste temporanee S e H per l'inserimento ordinato dei nodi e la tecnica del *flag* per la marcatura dei nodi già visitati. In particolare la lista temporanea S viene utilizzata per la memorizzazione in sequenza dei fratelli destri del nodo n, mentre la lista H viene utilizzata per la memorizzazione *document-order* dei *following-sibling* dei nodi della lista C. I nodi n' elaborati vengono inseriti in coda alla lista L se la loro posizione precede i nodi ancora da elaborare, altrimenti tali nodi vengono memorizzati temporaneamente nella lista S (righe 9-16). L'inserimento nella lista L avviene dopo l'inserimento dei nodi memorizzati nella lista H che precedono il nodo attuale (righe 12-14). Ad ogni elaborazione di una sequenza di *following-sibling* per ogni nodo $n \in C$ la lista S viene memorizzata all'inizio della lista H poichè nella lista

S sono presenti nodi che precedono sicuramente i nodi già presenti nella lista H, si tenga presente che la scansione degli elementi nella lista C è sempre *document-order*. Alla fine dell'algoritmo è necessario verificare che siano presenti ancora dei nodi nella lista H ed eventualmente aggiungerli alla fine della lista L, quest'operazione viene eseguita attraverso la funzione *AddListAfter(L,H)*. Attravero queste tecniche basate sull'utilizzo delle liste temporanee S e H il risultato, rappresentato dalla lista L, sarà sempre un insieme ordinato di nodi (*document-order*).

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè ogni nodo è visitato al più una sola volta (grazie all'utilizzo del *flag*).

La presenza dei due cicli *while* annidati (righe 12-14 e 3-22) non genera una complessità quadratica $O(n^2)$ poichè la somma gli elementi presenti nella lista C e nella lista H risulta essere, al più, pari a $O(n)$.

Elaborazione dell'asse preceding::a

```
preceding (C,a)
1   L ← NewList();
2   n ← Last(C);
3   while n ≠ NULL do
4       n' ← left-sibling(n,element);
5       S ← NewList();
6       while n' ≠ NULL do
7           if (tag(n')=a) or (a='*') then
8               AddAfter(S,n');
9           endif
10          flag(n') ← CONT;
11          AddAllDescendant (S,n',a);
12          n' ← left-sibling(n',element);
13      endw
14      AddListBefore(L,S);
15      n ← parent-node(n,element);
16  endw
17  return L;
```

Per l'elaborazione dell'asse **preceding::a** su una lista C di nodi ordinata (*document order*) è sufficiente effettuare il calcolo solo sull'ultimo elemento della lista C

(riga 2 dell'algoritmo). Questo perchè i nodi che precedono un insieme ordinato di nodi C sono compresi nei nodi che precedono l'ultimo nodo dell'insieme C.

In questo caso si utilizzata una lista temporanea S che simula una struttura dati di tipo *stack*, poichè ad ogni elaborazione del nodo fratello, alla sinistra del nodo attuale, l'inserimento avviene all'inizio della lista L (riga 14).

Anche in questo algoritmo si utilizza la funzione *AddAllDescendant* (riga 11) poichè il calcolo del *preceding* di un singolo nodo non produce sovrapposizioni, quindi l'utilizzo del *flag* risulta superfluo.

Come è possibile notare quest'algoritmo è molto simile all'algoritmo **following::a** proprio a causa della simmetria dei due assi; infatti following^{-1} = preceding.

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a *O(n)* poichè ogni nodo è visitato al più una sola volta.

Elaborazione dell'asse preceding-sibling::a

preceding-sibling (C,a)

```
1   L ← NewList();
2   H ← NewList();
3   while not(empty(C)) do
4       S ← NewList();
5       n ← DelLast(C);
6       n' ← left-sibling(n,element);
7       while (n' ≠ NULL) and (flag(n') < CONT) do
8           if (tag(n')=a) or (a='*') then
9               if (not(empty(C)) and pre(Last(C)) > pre(n') then
10                  AddBefore(S,n');
11              else
12                  while not(empty(H)) and pre(Last(H)) > pre(n') do
13                      AddBefore(L,DelLast(H));
14                  endw
15                  AddBefore(L,n');
16              endif
17          endif
18          flag(n') ← CONT;
19          n' ← left-sibling(n',element);
20      endw
21      AddListAfter (H,S);
22  endw
23  AddListBefore(L,H);
24  return L;
```

Come è possibile notare quest'algoritmo è la versione speculare dell'algoritmo **following-sibling::a**. In questo algoritmo la lista C viene scandita a partire dalla fine (riga 5) e gli inserimenti nella lista temporanea S e nella lista L avvengono all'inizio anzichè alla fine (righe 10,13,15). Si noti l'utilizzo dell'operatore *pre()* anzichè *post()* all'interno dell'istruzione condizionale nella riga 9 dell'algoritmo.

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè ogni nodo è visitato al più una sola volta (grazie all'utilizzo del *flag*).

Elaborazione degli assi next::a e next-sibling::a

Per poter calcolare l'asse **next::a** su un insieme C di nodi ordinati utilizziamo la funzione *Next-node(n)* che restituisce il nodo successivo, secondo l'ordine del documento XML, del nodo n; nel caso in cui il successore non esista la funzione restituisce il valore NULL.

Next-node (n)

```
1   n' ← right-sibling(n,element);
2   while (n' = NULL) and (n ≠ NULL) and (flag(n) < CONT) do
3       n ← parent-node(n,element);
4       flag(n) ← CONT;
5       n' ← right-sibling(n,element);
6   endw
7   return n';
```

Di seguito è riportato l'algoritmo per il calcolo degli assi **next::a** e **next-sibling::a**.

next (C,a,sibling)

```
1   L ← NewList();
2   S ← NewList();
3   while not(empty(C)) do
4       n ← DelFirst(C);
5       if sibling then
6           n' ← right-sibling(n,element);
7       else
8           n' ← Next-node(n);
9       endif
10      if (n' ≠ NULL) and (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
11          if not(empty(C)) and (post(First(C)) < post(n')) then
12              AddBefore(S,n');
13          else
14              while not(empty(S)) and (pre(First(S)) < pre(n')) do
15                  AddAfter(L,DelFirst(S));
16              endw
17              AddAfter(L,n');
18          endif
19          flag(n') ← CONT;
20      endif
21  endw
22  AddListAfter(L,S);
23  return L;
```

Per calcolare l'asse **next::a** è necessario richiamare la funzione con il parametro *sibling* impostato a *false* (*next(C,a,false)*), altrimenti per il calcolo dell'asse **next-sibling::a** è necessario impostare il parametro *sibling* a *true* (*next(C,a,true)*).

Quest'algoritmo utilizza una lista S che simula una struttura dati di tipo stack per la memorizzazione temporanea dei nodi che seguono i successori dei nodi nella lista C. In pratica questa struttura dati viene utilizzata per riordinare di volta in volta i successori dei nodi nella lista C.

Questa ulteriore artificiosità dell'algoritmo deriva dal fatto che possono esistere due nodi $n,n' \in C$ con $n < n'$ tali che *Next-node(n) > Next-node(n')* (oppure nel caso di **next-sibling::a** *right-sibling(n,element) > right-sibling(n',element)*) ed è quindi necessario operare un ordinamento in itinere.

Se indichiamo con n la cardinalità di tutti i nodi elemento dell'albero XML l'algoritmo appena descritto avrà, al più, complessità computazionale pari a $O(n)$ poichè ogni nodo è visitato al più una sola volta (grazie all'utilizzo del *flag* all'interno della funzione *Next-node(n)* e nel controllo di riga 10 dell'algoritmo).

La presenza delle due liste S e C e dei due cicli *while* annidati (righe 3-21 e 14-16) non genera una complessità computazionale di tipo quadratico $O(n^2)$ poichè la somma delle dimensioni della lista C e della lista S risulta essere, al più, pari a $O(n)$; nella lista L viene inserito un elemento proveniente dalla lista C o dalla lista S al più $O(|C|)$ volte.

L'algoritmo appena presentato è molto simile all'algoritmo utilizzato per l'elaborazione dell'asse *following-sibling* soprattutto per quel che riguarda l'utilizzo della lista temporanea S e delle strutture condizionali presenti nelle righe 10-20 dell'algoritmo.

Elaborazione dell'asse previous::a e previous-sibling::a

Per poter calcolare l'asse **previous::a** su un insieme di nodi ordinati C utilizziamo la funzione *Previous-node(n)* che restituisce il nodo precedente, secondo l'ordine del documento XML, del nodo n; nel caso in cui il predecessore non esista la funzione restituisce il valore NULL.

Previous-node (n)

```
1   n' ← left-sibling(n,element);
2   while (n' = NULL) and (n ≠ NULL) and (flag(n) < CONT) do
3       n ← parent-node(n,element);
4       flag(n) ← CONT;
5       n' ← left-sibling(n,element);
6   endw
7   return n';
```

Di seguito è riportato l'algoritmo per il calcolo degli assi **previous::a** e **previous-sibling::a**.

previous (C,a,sibling)

```
1   L ← NewList();
2   S ← NewList();
3   while not(empty(C)) do
4       n ← DelLast(C);
5       if sibling then
6           n' ← left-sibling(n,element);
7       else
8           n' ← Previous-node(n);
9       endif
10      if (n' ≠ NULL) and (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
11          if not(empty(C)) and (pre(Last(C)) > pre(n')) then
12              AddBefore(S,n');
13          else
14              while not(empty(S)) and (pre(Last(S)) > pre(n')) do
15                  AddAfter(L,DelLast(S));
16              endw
17              AddBefore(L,n');
18          endif
19          flag(n') ← CONT;
20      endif
21  endw
22  AddListBefore(L,S);
23  return L;
```

Come è possibile notare quest'algoritmo è molto simile all'algoritmo per il calcolo degli assi **next::a** e *next-sibling::a* proprio a causa della simmetria dei due assi; infatti $\text{next}^{-1} = \text{previous}$ e $\text{next-sibling}^{-1} = \text{previous-sibling}$. L'algoritmo per il calcolo degli assi *previous* e *previous-sibling* risulta essere speculare rispetto all'algoritmo per gli

assi *next* e *next-sibling*, così come l'algoritmo *preceding-sibling* è speculare rispetto a *following-sibling*. Anche in questo caso per poter elaborare l'asse **previous::a** si utilizza il parametro *sibling=false* e per l'asse **previous-sibling::a** il parametro *sibling=true*. Come per l'algoritmo **next(C,a,sibling)** anche questo ha complessità computazionale al più *O(n)* dove n è la cardinalità di tutti i nodi elemento dell'albero XML.

Elaborazione dell'asse id::a

```
id (C,a)
1   L ← NewList();
2   for i=1 to | T | do A[i] ← NULL;
3   while not(empty(C)) do
4       n ← DelFirst(C);
5       if type(n) ∈ { idref, idrefs } then
6           value[ ] ← split(S(n));
7           foreach v ∈ value[ ] do
8               n' ← hash(v);
9               if (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
10                  A[pre(n')] ← n';
11                  flag(n') ← CONT;
12              endif
13          endfch
14      endif
15  endw
16  for i=1 to | T | do
17      if A[i] ≠ NULL then
18          AddList(L,A[i]);
19      endif
20  endfor
21  return L;
```

Per l'elaborazione dell'asse **id::a** si utilizza la funzione *hash('s')* che restituisce, in media in tempo costante *O(1)*, il nodo dell'albero XML con attributo di tipo *id='s'*. Questa funzione *hash()* è implementata attraverso l'utilizzo di una tabella *hash* elaborata in fase di costruzione dell'albero XML.

Un'altra funzione utilizzata all'interno di questo algoritmo è *S(n)*, già introdotta nella definizione di Albero XML (vedi Paragrafo 3.3), che restituisce il valore della

stringa associata al nodo n.

Per la suddivisione dei valori id presenti all'interno della stringa *S(n)* si utilizza la funzione *split()*, anch'essa già introdotta nella definizione della Semantica EXPath (vedi Paragrafo 3.4). Il risultato della funzione *split()* viene memorizzato all'interno di un vettore denominato value (riga 5 dell'algoritmo).

Per poter restituire una lista ordinata di nodi si utilizza un vettore A avente lunghezza pari al numero dei nodi dell'albero T del documento XML. In questo vettore A vengono memorizzati di volta in volta i nodi *n'* ottenuti dall'elaborazione della funzione *hash()*. Il nodo *n'* viene inserito nel vettore A nella posizione *pre(n')* (riga 10 dell'algoritmo); in questo modo alla fine dell'elaborazione è possibile ottenere una lista ordinata di nodi effettuando semplice una scansione completa del vettore A a partire dalla prima posizione (righe 16-20 dell'algoritmo).

Prima di effettuare l'elaborazione il vettore A viene inizializzato con valori nulli (NULL) nella riga 2 dell'algoritmo.

Per poter determinare la complessità computazionale di quest'algoritmo nel caso peggiore dobbiamo considerare che la funzione *S(n)* (riga 6), per i nodi n di tipo attributo, ha complessità costante *O(1)* e che la funzione *hash(v)* (riga 8) ha complessità, nel caso peggiore, pari a *O(n)*; utilizzando comunque una *buona funzione hash* la complessità media risulta constante *O(1)*.

La complessità dell'algoritmo risulta essere al più *O(n)* per l'inizializzazione del vettore A, *O(n)* per l'elaborazione dell'asse *id* secondo le indicazioni date in precedenza e *O(n)* per la scansione finale del vettore A per la creazione della lista L ordinata: sommando questi valori si ottiene *O(3n) = O(n)*.

Esiste un caso limite nel quale ogni nodo del documento XML ha un attributo di tipo IDREFS con *O(n)* elementi. In questo caso la complessità computazionale dell'elaborazione di un'interrogazione del tipo `id(/descendant::*/attribute::*)` avrebbe complessità $O(n^2)$; ma n^2 nel nostro modello di albero è proprio la dimen-

sione del documento XML poichè per noi un nodo di tipo IDREFS di dimensione n è costituito da un singolo nodo e non da n nodi di tipo IDREF. Quindi anche in questo caso la complessità dell'elaborazione dell'asse *id* ha complessità al più lineare nella dimensione dell'albero XML.

Elaborazione dell'asse id^{-1}::a

id'(C,a)

```
1   L ← NewList();
2   for i=1 to | T | do A[i] ← NULL;
3   while not(empty(C)) do
4       n ← DelFirst(C);
5       if type(n) = id then
6           L' ← NewList();
7           L' ← hash'(S(n));
8           while not(empty(L')) do
9               n' ← DelFirst(L');
10              if (flag(n') < CONT) and ((tag(n')=a) or (a='*')) then
11                  A[pre(n')] ← n';
12                  flag(n') ← CONT;
13              endif
14          endw
15      endif
16  endw
17  for i=1 to | T | do
18      if A[i] ≠ NULL then
19          AddList(L,A[i]);
20      endif
21  endfor
22  return L;
```

Quest'algoritmo è molto simile a quello presentato per l'elaborazione dell'asse **id::a**; l'unica differenza è costituita dalla presenza della funzione *hash'()* e della lista *L'*. La funzione *hash'('s1')* restituisce tutti i nodi dell'albero XML che hanno un attributo di tipo *idref* o *idrefs* con un valore *s2* tale che *s1* $\in$ *split(s2)* ossia tutti i nodi dell'albero XML che hanno un puntatore verso il nodo con *id* pari a *s1*.

Anche questa funzione viene implementata grazie ad una tabella hash costruita in

fase di costruzione, in memoria, dell'albero XML. A differenza della funzione *hash()* la funzione *hash'(s)* restituisce una lista di nodi che rappresentano proprio i nodi che hanno un puntatore verso il nodo con *id* pari a *s*. La lista *L'* è una lista temporanea utilizzata per la memorizzazione del risultato *hash'(S(n))* (riga 7 dell'algoritmo).

Anche in questo caso, come per l'asse **id::a**, viene utilizzato un vettore A per restituire in maniera ordinata, secondo i valori pre(), i nodi elaborati dall'asse $\mathbf{id}^{-1}$.

La complessità computazionale di questo algoritmo è la stessa dell'algoritmo per l'elaborazione dell'asse **id::a** ossia è *O(n)*. Anche in questo caso valgono le stesse considerazioni nel caso limite di documenti XML con tutti i nodi collegati con tutti gli altri nodi del documento XML tramite un attributo di tipo IDREFS con *O(n)* elementi.

Algoritmo Evaluate(axis,a,C) per l'elaborazione degli assi

Evaluate (axis,a,C)

```
1   switch axis do
2       case self : C ← self(C,a);
3       case child : C ← child(C,a);
4       case parent : C ← parent(C,a);
5       case self-attribute : C ← self-attribute(C,a);
6       case attribute : C ← attribute(C,a);
7       case parent-attribute : C ← parent-attribute(C,a);
8       case descendant : C ← descendant(C,a,false);
9       case descendant-or-self : C ← descendant(C,a,true);
10      case ancestor : C ← ancestor(C,a,false);
11      case ancestor-or-self : C ← ancestor(C,a,true);
12      case following : C ← following(C,a);
13      case following-sibling : C ← following-sibling(C,a);
14      case preceding : C ← preceding(C,a);
15      case preceding-sibling : C ← preceding-sibling(C,a);
16      case next : C ← next(C,a,false);
17      case next-sibling : C ← next-sibling(C,a,true);
18      case previous : C ← previous(C,a,false);
19      case previous-sibling : C ← previous-sibling(C,a,true);
20      case id : C ← id(C,a);
21      case id^-1 : C ← id'(C,a);
22  endsw
23  CONT ← CONT + 1;
24  if CONT = MAX then
25      ResetTreeFlag();
26      CONT ← 1;
27  endif
28  return C;
```

L'algoritmo **Evaluate(axis,a,C)** consente di elaborare un asse generico *axis::a* su un insieme ordinato C di nodi dell'albero XML attraverso l'utilizzo di tutti gli algoritmi sugli assi presentati in questo Paragrafo. Si noti la presenza dell'incremento del contatore CONT e del controllo sul suo valore massimo ammissibile MAX che consente di riazzerare il campo *flag* dei nodi dell'albero XML (righe 23-27 dell'algoritmo).

Dal momento che abbiamo dimostrato che l'elaborazione di qualsiasi asse ha complessità computazionale al più $O(n)$ con n che rappresenta la dimensione del docu-

mento XML, la complessità dell'algoritmo *Evaluate(axis,a,C)* risulta essere *O(n)*.

3.6.5 Elaborazione dei filtri

Per poter elaborare i filtri presenti nel linguaggio EXPath in maniera efficiente utilizziamo una tecnica introdotta in [1]. I filtri vengono specificati nel linguaggio EXPath attraverso l'utilizzo della sintassi *q[p]* dove $q \in \{\text{id('s'), axis::a}\}$ e p è un percorso (path). Tramite l'utilizzo di una funzione $\tau(p)$: EXPath $\mapsto$ EXPath che restituisce il percorso inverso di p è possibile elaborare l'interrogazione *q[p]* intersecando il risultato di $f(T, q, C)$ con il risultato di $f(T, \tau(p), N)$. In questo modo l'interrogazione *q[p]* viene trasformata in un'interrogazione EXPath equivalente senza l'utilizzo dell'operatore filtro *[p]*. In particolare è possibile dimostrare la seguente proposizione.

Proposizione 1 (Elaborazione dei filtri EXPath)**.** *Sia q[p] un'interrogazione EXPath dove $q \in \{$id('s'), axis::a$\}$ e p è un percorso (path). Sia T l'albero XML ed N l'insieme dei nodi dell'albero, è possibile riscrivere quest'interrogazione in una forma equivalente senza l'utilizzo dell'operatore filtro [p] nel modo seguente:*

$$f(T,q[p],C) = f(T,q,C) \cap f(T,\tau(p),N)$$

tramite l'introduzione di una funzione di traduzione $\tau(p)$: EXPath $\mapsto$ EXPath dove $\tau(p)$ è il percorso inverso di p, ossia se n, n' $\in$ N allora n' p n $\Leftrightarrow$ n $\tau(p)$ n'.

Dimostrazione. Sia *q[p]* un'interrogazione EXPath, la semantica di *q[p]* è definita nel modo seguente $f(T, q[p], C) = \{n \in f(T, q, C) \mid \exists m \in N.\ m \in f(T, p, \{n\})\}$ ossia i nodi n ottenuti dall'elaborazione del percorso q per i quali l'applicazione del percorso p produce un insieme di nodi m non nullo. Indichiamo con A l'insieme $f(T, q[p], C)$ e con B l'insieme $f(T, q, C) \cap f(T, \tau(p), N)$. Vogliamo dimostrare che se $x \in A \Rightarrow x \in B$ e viceversa.

Se $x \in A$ allora $x \in f(T, q, C)$ e $\exists m \in N$ tale che $m \in f(T, p, \{x\})$. Se indichiamo con $\tau(p)$ il percorso inverso di p allora se $m\ p\ x$ allora $x\ \tau(p)\ m$ ossia se $m \in f(T, p, \{x\}) \Rightarrow x \in f(T, \tau(p), \{m\})$. Risulta che $x \in f(T, q, C)$ e $x \in f(T, \tau(p), \{m\}) \subset f(T, \tau(p), N)$ quindi $x \in f(T, q, C) \cap f(T, \tau(p), N) \Rightarrow x \in B$.

Se $x \in B$ allora $x \in f(T, q, C)$ e $x \in f(T, \tau(p), N) \Rightarrow x \in f(T, \tau(p), \{m\})$ con $m \in N \Rightarrow m \in f(T, p, \{x\})$ ossia $x \in f(T, q, C)$ e $\exists m \in N.\ m \in f(T, p, \{x\}) \Rightarrow x \in A$. □

Così facendo, se l'elaborazione dei percorsi q e $\tau(p)$ può essere risolta in maniera efficiente, anche l'interrogazione di istruzioni EXPath contenenti filtri può essere risolta in maniera efficiente poichè l'intersezione di due insiemi ordinati con n_1 e n_2 elementi ha una complessità computazionale $O(n_1 + n_2)$, ossia è al più lineare nella somma degli elementi degli insiemi.

Nell'elaborazione dei filtri EXPath del metodo precedente è presente la valutazione della funzione $f(T, \tau(p), N)$ su tutto l'insieme N dei nodi dell'albero XML. La presenza dell'insieme N in fase di elaborazione del filtro può rappresentare un problema, in fase di implementazione, a causa dello spazio di memorizzazione di quest'insieme che può assumere dimensioni elevate soprattutto in presenza di documenti XML di grandi dimensioni. Per ovviare a questo problema si è pensato ad un metodo alternativo di elaborazione dei filtri EXPath senza l'utilizzo dell'insieme N.

Proposizione 2 (Metodo alternativo di elaborazione dei filtri)**.** *Sia q[p] un'interrogazione in EXPath dove $q \in \{$ id('s'), axis::a$\}$ e p è un percorso (path). Sia T l'albero XML, $C \subseteq N$ dove N è l'insieme di tutti i nodi dell'albero T e $\tau(p)$ il percorso inverso di p, allora risulta:*

$$f(T, q[p], C) = f(T, q, C) \cap f(T, \tau(p), f(T, p, f(T, q, C)))$$

Dimostrazione. La semantica di $q[p]$ risulta essere:

$$f(T, q[p], C) = \{n \in f(T, q, C) \mid \exists m \in N.\ m \in f(T, p, \{n\})\}$$

Indichiamo con A l'insieme $f(T, q[p], C)$ e con B l'insieme $f(T, q, C) \cap f(T, \tau(p), f(T, p, f(T, q, C)))$. Vogliamo dimostrare che se $x \in A \Rightarrow x \in B$ e viceversa.

Se $x \in A$ allora $x \in f(T, q, C)$ e $\exists m \in N$ tale che $m \in f(T, p, \{x\}) \Rightarrow m \in f(T, p, f(T, q, C))$. Dal momento che $\tau(p) = p^{-1}$ risulta che se $m\ p\ x$ allora $x\ \tau(p)\ m \Rightarrow x \in f(T, \tau(p), \{m\}) \Rightarrow x \in f(T, \tau(p), f(T, p, f(T, q, C))) \Rightarrow x \in B$.

Se $x \in B$ allora $x \in f(T, q, C)$ e $x \in f(T, \tau(p), f(T, p, f(T, q, C))) \Rightarrow x \in f(T, q, C)$ e $x \in f(T, \tau(p), \{m\})$ con $m \in f(T, p, f(T, q, C)) \Rightarrow x \in f(T, q, C)$ e $m \in f(T, p, \{x\})$ e $m \in f(T, p, f(T, q, C)) \Rightarrow x \in f(T, q, C)$ e $\exists m \in N$ con $m \in f(T, p, \{x\}) \Rightarrow x \in A$. □

Per poter completare l'analisi di questo metodo alternativo (Proposizione 2) per l'elaborazione dei filtri EXPath dobbiamo calcolarne la complessità computazionale e confrontarla con quella del metodo originale proposto in [1] (Proposizione 1).

Indichiamo con $n = |\mathrm{N}|$, con k la lunghezza del percorso p e con k' la lunghezza di $\tau(p)$. Considerando che $\tau(p)$ tende, al più, a raddoppiare la lunghezza del percorso p, possiamo ipotizzare che $k' = 2k$ (ciò si può dedurre osservando la costruzione della funzione $\tau(p)$ riportata nella Definizione 10).

Calcoliamo la complessità computazionale del secondo metodo (Proposizione 2) ipotizzando che la complessità computazionale di $f(T, p, C)$ sia lineare del tipo $O(k \cdot n)$, dove p è un percorso generico. La complessità computazionale di $f(T, q, C) \cap f(T, \tau(p), f(T, p, f(T, q, C)))$ risulta essere:

$$O(n + n + k \cdot n + k' \cdot n) = O(2n + 3k \cdot n) = O(k \cdot n)$$

poichè la complessità computazionale dell'intersezione di due insiemi ordinati con n_1 e n_2 elementi è $O(n_1 + n_2)$.

La complessità computazionale di $f(T, q, C) \cap f(T, \tau(p), N)$ del primo metodo (Proposizione 1) considerando le stesse ipotesi risulta essere:

$$O(n + k' \cdot n) = O(n + 2k \cdot n) = O(k \cdot n)$$

In termini di complessità computazionale i due metodi risultano essere equivalenti.

A questo punto rimane solo da definire la funzione $\tau(p)$ come la funzione che restiruisce il percorso inverso di p ossia se dati $x, y \in N$ e $x \ p \ y \Rightarrow y \ \tau(p) \ x$.

Definizione 10 (Percorso inverso)**.** *Sia p un percorso (path) in EXPath. La funzione* $\tau(p) : EXPath \mapsto EXPath$ *che restituisce il percorso inverso di p è definita nel modo seguente:*

$$\tau(p) = \begin{cases} \tau(path_2)/\tau(path_1) & se \quad p= path_1/path_2 \\ \tau(path)/id('s') & se \quad p= id('s')[path] \\ \tau(path)/id('s_1')='s_2' & se \quad p= id('s_1')[path]='s_2' \\ self::a/axis^{-1}::* & se \quad p= axis::a \ e \ axis \notin \{ self, \\ & \quad attribute, \ self\text{-}attribute \ \} \\ axis^{-1}::a & se \quad p= axis::a \ e \ axis \in \{ self, \\ & \quad self\text{-}attribute \ \} \\ self\text{-}attribute::a/parent\text{-}attribute::* & se \quad p= attribute::a \\ \tau(path)/\tau(axis::a) & se \quad p= axis::a[path] \\ self::a='s'/axis^{-1}::* & se \quad p= axis::a='s' \ e \ axis \notin \{ self, \\ & \quad attribute, \ self\text{-}attribute \ \} \\ axis^{-1}::a='s' & se \quad p= axis::a='s' \ e \ axis \in \{ self, \\ & \quad self\text{-}attribute \ \} \\ self\text{-}attribute::a='s'/parent\text{-}attribute::* & se \quad p= attribute::a='s' \\ \tau(path)/\tau(axis::a='s') & se \quad p= axis::a[path]='s' \\ \tau(path_1) \ and \ \tau(path_2) & se \quad p= path_1 \ and \ path_2 \\ \tau(path_1) \ or \ \tau(path_2) & se \quad p= path_1 \ or \ path_2 \\ not(\tau(path)) & se \quad p= not(path) \\ p & altrimenti \end{cases}$$

con $a \in \Sigma \cup \{*\}$ *e* $s, \ s_1, \ s_2 \in String$*.*

Di seguito è riportato l'algoritmo **Inverse(path)** che implementa la funzione $\tau(p)$ appena definita.

Inverse (path)

```
1  if path = path1/path2 then
2      R ← Inverse(path2)/Inverse(path1);
3  else if path = id('s')[path] then
4      R ← Inverse(path)/id('s');
5  else if path = id('s1')[path]='s2' then
6      R ← Inverse(path)/id('s1')='s2';
7  else if (path = axis::a) and (axis ∉ {attribute, self, self-attribute}) then
8      R ← self::a/axis^-1::*;
9  else if (path = axis::a) and (axis ∈ {self, self-attribute}) then
10     R ← axis^-1::a;
11 else if path = attribute::a then
12     R ← self-attribute::a/parent-attribute::*;
13 else if path = axis::a[path] then
14     R ← Inverse(path)/Inverse(axis::a);
15 else if (path = axis::a='s') and (axis ∉ {attribute, self, self-attribute}) then
16     R ← self::a='s'/axis^-1::*;
17 else if (path = axis::a='s') and (axis ∈ {self, self-attribute}) then
18     R ← axis^-1::a='s';
19 else if (path = attribute::a='s') then
20     R ← self-attribute::a='s'/parent-attribute::*;
21 else if (path = axis::a[path]='s') then
22     R ← Inverse(path)/Inverse(axis::a='s');
23 else if (path = path1 and path2) then
24     R ← Inverse(path1) and Inverse(path2);
25 else if (path = path1 or path2) then
26     R ← Inverse(path1) or Inverse(path2);
27 else if (path = not(path)) then
28     R ← not(Inverse(path));
29 else
30     R ← path;
31 return R;
```

Se indichiamo con k la lunghezza del percorso *path*, la complessità computazionale dell'algoritmo **Inverse(path)** è $O(k)$, ossia è lineare nella dimensione dell'input.

3.6.6 L'algoritmo di valutazione di una query in EXPath

EXPath (T,q,C)

```
1   if not(empty(C)) then
2       if q = /path then
3           C ← EXPath(T,path,{root});
4       else if q = q1/q2 then
5           C ← EXPath(T,q2,EXPath(T,q1,C));
6       else if q = id('s') then
7           C ← hash('s');
8       else if q = id('s')[path] then
9           C' ← hash('s');
10          C ← C' ∩ EXPath(T,Inverse(path),EXPath(T,path,C'));
11      else if q = id('s1')='s2' then
12          C ← Equal(T,hash('s1'),'s2');
13      else if q = id('s1')[path]='s2' then
14          C ← Equal(T,EXPath(T,id('s1')[path],C),'s2');
15      else if q = path1 and path2 then
16          C ← EXPath(T,path1,C) ∩ EXPath(T,path2,C);
17      else if q = path1 or path2 then
18          C ← EXPath(T,path1,C) ∪ EXPath(T,path2,C);
19      else if q = not(path) then
20          C ← N \ EXPath(T,path,C);
21      else if q = axis::a then
22          C ← Evaluate(axis,a,C);
23      else if q = axis::a[path] then
24          C' ← Evaluate(axis,a,C);
25          C ← C' ∩ EXPath(T,Inverse(path),EXPath(T,path,C'));
26      else if q = axis::a='s' then
27          C ← Equal(T,Evaluate(axis,a,C),'s');
28      else if q = axis::a[path]='s' then
29          C ← Equal(T,EXPath(T,axis::a[path],C),'s');
30  endif
31  return C;
```

Calcoliamo la complessità computazionale di questo algoritmo nel caso peggiore. Indichiamo con $k = |\,\mathrm{q}|$ la lunghezza della query EXPath e con n la dimensione del documento XML. La complessità computazionale per la risoluzione di interrogazioni EXPath risulta essere al più $O(k \cdot n)$. In particolare abbiamo dimostrato la seguente

proposizione.

Proposizione 3 (Complessità dell'elaborazione di un'interrogazione EXPath)**.** *Sia q un'interrogazione EXPath, D un documento XML e T la rappresentazione ad albero del documento D, allora l'elaborazione dell'interrogazione q su un insieme ordinato di nodi $C \subseteq T$, ossia l'elaborazione di f(T,q,C) (vedi definizione 7), può essere risolta con complessità computazionale lineare $O(k \cdot n)$ dove $k = |q|$ è la lunghezza della query EXPath ed $n = |D|$ è la dimensione del documento XML.*

3.6.7 Proposta per un nuovo algoritmo di valutazione di una query in EXPath

Un'idea alternativa per l'elaborazione di un'interrogazione EXPath è quella di far cadere l'ipotesi dell'insieme ordinato ad ogni step dell'elaborazione e mantenere l'ordinamento solo alla fine dell'elaborazione di tutta l'interrogazione.

Con questa ipotesi vengono meno le informazioni derivanti dall'ordinamento dei nodi da elaborare necessarie per poter utilizzare le tecniche *pre/post* esposte in questo paragrafo.

La proposta è quella di utilizzare solo la tecnica del *flag* numerico per risolvere tutti gli assi EXPath ipotizzando quindi di non dover restituire un insieme ordinato di nodi ad ogni *step* dell'elaborazione.

Con questa ipotesi gli algoritmi per l'elaborazione degli assi si semplificano notevolmente perchè è sufficiente inserire esclusivamente il controllo sul *flag* di un nodo e procedere all'elaborazione dei nodi successivi; la complessità sarà al più $O(n)$ poichè i nodi verranno visitati al più una volta sola.

Alla fine dell'elaborazione dell'interrogazione EXPath è necessario effettuare l'ordinamento dell'insieme ottenuto, perchè in output vogliamo sempre un insieme di nodi *document order*. La complessità di questo ordinamento sarà al più $O(n)$ poichè verrà utilizzato lo stesso sistema presentato per l'elaborazione degli assi **id::a** e $\mathbf{id}^{-1}$**::a**, ossia l'utilizzo di un vettore A di $O(n)$ elementi.

In pratica al termine dell'elaborazione di tutti gli *step* dell'interrogazione EXPath è sufficiente inserire i nodi elaborati nel vettore A ed effettuare una scansione del vettore per ottenere la lista ordinata dei nodi risultato.

Questo nuovo algoritmo per la risoluzione di interrogazioni EXPath potrebbe essere confrontato con quello precedente che utilizza la tecnica del *pre/post* per valutare le performance operative di queste due tecniche.

In particolare il confronto potrebbe essere interessante per valutare la reale utilità delle informazioni *pre/post* legate ad ogni singolo nodo.

3.6.8 La funzione di traduzione da SXPath a EXPath

SX2EXPath (q)

```
1  if q = id(path) then
2      Q ← SX2EXPath(path)/id::∗;
3  else if (q = axis::a[filter]) and ((axis ∉ {attribute, self-attribute}) or
   (filter≠parent::b(/path))) then
4      Q ← SX2EXPath(axis::a)[SX2EXPath(filter)];
5  else if (q = axis::a[filter]) and (axis ∈ {attribute, self-attribute}) and
   (filter=parent::b(/path))) then
6      Q ← SX2EXPath(axis::a)[parent-attribute::b(/SX2EXPath(path))];
7  else if q = filter='s' then
8      Q ← SX2EXPath(filter)='s';
9  else if q = filter_1 and filter_2 then
10     Q ← SX2EXPath(filter_1) and SX2EXPath(filter_2);
11 else if q = filter_1 or filter_2 then
12     Q ← SX2EXPath(filter_1) or SX2EXPath(filter_2);
13 else if q = not(filter) then
14     Q ← not(SX2EXPath(filter));
15 else if q = /q_1/.../q_n then
16     for i=1 to n do
17         if (i > 1) and (q_i=parent::a) then
18             if q_{i-1}=attribute::b then
19                 Q_i ← parent-attribute::a;
20             else
21                 Q_i ← q_i;
22             endif
23         else
24             Q_i ← SX2EXPath(q_i);
25         endif
26     endfor
27     Q ← /Q_1/.../Q_n;
28 else
29     Q ← q;
30 endif
31 return Q;
```

Se indichiamo con k la lunghezza del percorso q, la complessità computazionale dell'algoritmo **SX2EXPath(q)** è $O(k)$, ossia è lineare nella dimensione dell'input.

3.6.9 L'algoritmo di valutazione di una query in SXPath

Per poter elaborare un'interrogazione in SXPath è sufficiente utilizzare gli algoritmi presentati in questo capitolo e precisamente l'algoritmo di traduzione *SX2EXPath()* e l'algoritmo di elaborazione *EXPath()*.

SXPath (T,q,C)

```
1  q' ← SX2EXPath (q);
2  Q ← EXPath (T,q',C);
3  return Q;
```

Sia T la rappresentazione ad albero di un documento XML, q un'interrogazione SXPath e C l'insieme dei nodi da elaborare; se indichiamo con con $k = |\,\mathrm{q}\,|$ la lunghezza dell'interrogazione SXPath e con n la dimensione del documento XML l'algoritmo *SXPath(T,q,C)* ha complessità computazionale pari a $O(k) + O(k \cdot n)$ ossia $O(k \cdot n)$.

L'algoritmo **SXPath(T,q,C)** appena presentato ha dunque la stessa complessità computazionale dell'algoritmo *EXPath(T,q,C)*.

Il risultato appena ottenuto può essere generalizzato e formalizzato in questa proposizione.

Proposizione 4 (Elaborazione di un'interrogazione SXPath)**.** *Sia q un'interrogazione in SXPath allora esiste un'interrogazione q' in EXPath tale che:*

1. *la semantica di q è equivalente alla semantica di q';*
2. *q' = $\phi(q)$, dove ϕ è la funzione di traduzione da SXPath a EXPath;*
3. *q' è ottenuta da q in tempo lineare;*
4. *l'elaborazione di q ha la stessa complessità computazionale dell'elaborazione di q'.*

Dimostrazione. I punti 1 e 2 discendono dalla definizione di semantica SXPath e dall'utilizzo della funzione di traduzione ϕ (Definizione 8). I punti 3 e 4 sono dimostrati rispettivamente dal calcolo della complessità degli algoritmi *SX2EXPath(q)* e *SXPath(T,q,C)*. □

A conclusione del capitolo possiamo affermare di essere riusciti a dimostrare che è possibile risolvere con complessità computazione al più lineare $O(k \cdot n)$ nella dimensione n del documento XML e nella dimensione k della query, interrogazioni EXPath e SXPath; abbiamo quindi esteso il risultato ottenuto da Georg Gottlob, Christoph Koch e Reinhard Pichler in [1, 14] anche al caso di documenti XML con attributi e riferimenti.

In realtà si è riusciti anche a dimostrare che è possibile risolvere con complessità lineare non solo interrogazioni Core XPath e quindi SXPath su documenti XML con attributi e riferimenti ma anche interrogazioni EXPath, ottenendo così un risultato più ampio.

Capitolo 4

Implementazione e test di efficienza

In questo Capitolo è presentata una prima implementazione dell'algoritmo EXPath discusso in precedenza. Tale implementazione, tuttavia, non è completa ed è stata realizzata solo per un sottoinsieme del linguaggio EXPath che utilizza tutti gli assi tranne id e id^{-1}, senza l'impiego dell'operatore filtro e senza gli operatori logici e l'operatore di uguaglianza. Inoltre allo stato attuale l'implementazione non gestisce l'eventuale DTD associata al documento XML.

Questa implementazione è stata realizzata per verificare la bontà dei risultati teorici ottenuti nel Capitolo 3 e per fornire un base solida di partenza per una futura implementazione di un engine completo per i linguaggi EXPath e SXPath.

4.1 L'implementazione in C

Per l'implementazione dell'engine EXPath si è scelto di utilizzare il linguaggio C standard ANSI in modo da ottenere un software veloce e facilmente trasportabile su diversi sistemi operativi. L'implementazione è avvenuta in ambiente Gnu/Linux utilizzando il compilatore *gcc* di Gnu Operation System Project[1] (Free Software Foundation[2]). Per la realizzazione dell'albero XML a partire da un documento XML

[1] http://www.gnu.org
[2] http://www.fsf.org

si è utilizzato il parser Expat[3] realizzato da James Clark e rilasciato con licenza GPL[4] (General Public License). Questo parser, rappresentato da una libreria C, consente di generare delle chiamate di funzioni (*callback*) ad ogni lettura di un elemento di un documento XML. Gli eventi gestiti dal parser che sono stati utilizzati sono: il rilevamento di un tag di tipo elemento, di un nodo attributo, di un nodo di tipo testo ed infine il rilevamento del tag di chiusura di un elemento. La costruzione dell'albero XML in memoria RAM (*memory tree*) è stata realizzata tramite questa gestione ad eventi della libreria Expat e tramite l'utilizzo di due stack per la memorizzazione temporanea dei nodi genitori (*up*) e fratelli sinistri (*left*). In particolare è stata implementata una libreria, denominata *memorytree.h*, contentente una funzione, denomiata *XmlTree()*, che genera l'albero XML in memoria RAM a partire da un documento XML, memorizzato nello standard input. Il memory-tree è rappresentato da una struttura di puntatori a partire da un nodo radice (*root*).

Ogni nodo dell'albero ha una struttura C di questo tipo:

```
struct node {
   tnode type;
   const char *tag;
   long  pre,post;
   int   flag;
   struct node *parent,*child,*right_sibling,*left_sibling;
};
```

dove $type \in \{all, element, text, attribute, ID, IDREF, IDREFS\}$, *tag* è un puntatore ad una stringa contenente il tag associato al nodo, *pre* e *post* sono i valori di *pre/post* del nodo, *flag* è il valore del flag numerico ed infine *parent*, *child*, *right_sibling* e *left_sibling* sono i puntatori al nodo padre, figlio, fratello sinistro e fratello destro del nodo attuale.

Lo spazio occupato in memoria da questa struttura ad albero è stato valutato essere circa il doppio della dimensioni del documento XML; ciò dipende dal fatto che per ogni nodo del *memory tree* vengono memorizzate numerose informazioni in più rispetto al documento XML, tutte necessarie per l'ottimizzazione degli algoritmi

[3] http://expat.sourceforge.net
[4] http://www.gnu.org/copyleft/gpl.html

di navigazione all'interno di questa struttura ad albero (ad esempio i campi *pre*, *post* e *flag* vengono utilizzati solo per migliorare le performance degli algoritmi di elaborazione delle query XPath). In Figura 4.1 sono riportati i valori sperimentali ottenuti dall'elaborazione di 11 file XML con dimensioni variabili da 0.166 Mb fino a 166.517 Mb generati tramite il benchmark XMark [19].

XMark factor	**Dimensione file XML**	**Num. di nodi**	**Dimensione memory tree**
0.001	0.116	3'657	0.226
0.002	0.212	6'527	0.409
0.004	0.468	14'870	0.921
0.008	0.909	28'723	1.783
0.016	1.891	58'845	3.681
0.032	3.751	116'336	7.292
0.064	7.303	229'516	14.291
0.128	15.044	467'004	29.268
0.256	29.887	925'786	58.077
0.512	59.489	1'850'559	115.841
1.000	116.517	3'613'248	226.549

Figura 4.1: Dimensione file XML e *memory tree* (in Mb)

La parte fondamentale dell'implementazione corrispondente agli algoritmi per la risoluzione di interrogazioni EXPath riportati nel Capitolo 3 è riportata nel listato `EXPath.c` dove è presente la funzione *EXPath()* che resituisce l'insieme ordinato dei nodi risultato.

Per una visione completa del listato `EXPath.c` e di tutti gli altri sorgenti C si rimanda all'Appendice. In particolare, in suddetta sezione, sono presenti i sorgenti dei file `EXPath.c` contenente l'engine EXPath ed i file `memorytree.h` e `memorytree.c` contenenti la libreria per la creazione del *memory tree* a partire da un documento XML.

4.2 L'utilizzo dell'engine EXPath

L'engine EXPath è stato implementato su un sistema Gnu/Linux[5] e consente di eseguire diverse operazioni a seconda dei parametri specificati nella riga di comando. L'engine EXPath viene eseguito come comando, in una *shell*, tramite la seguente sintassi:

```
$ ./EXPath [-q query] [-m] [-d] [-dpp] < file.xml
```

dove `-q` è il parametro che indica la `query` EXPath da elaborare, `-m` è un parametro che restituisce informazioni sul *memory tree* (in particolare il numero di nodi e la dimensione in byte dell'albero XML), `-d` e `-dpp` sono due parametri che consentono di ottenere in output una rappresentazione nel linguaggio Dot[6] del *memory tree*.

Il `file.xml` viene specificato nel comando `./EXPath` come valore dello standard input (`< file.xml`). Consideriamo, ad esempio, il seguente documento XML denominato `turing.xml`:

```
<?xml version="1.0"?>
<person code="123">
  text1
  <name age="42">Alan</name>
  text2
  <surname>Turing</surname>
</person>
```

Per eseguire una query EXPath, ad esempio `/descendant::surname`, è necessario utilizzare il seguente comando:

[5]Dal momento che l'engine EXPath è stato implementato in standard ANSI C può essere facilmente compilato anche su altri sistemi operativi come ad esempio Ms Windows.

[6]Il linguaggio Dot è un linguaggio del progetto GraphViz, originario dei laboratori AT&T Bells, utilizzato per la rappresentazione di grafi in generale. Per maggiori informazioni http://www.graphviz.org

```
$ ./EXPath -q /descendant::surname < turing.xml
<surname>Turing</surname>
```

Il risultato dell'elaborazione, anzichè essere visualizzato a video, può essere memorizzato in un file semplicemente reindirizzando lo standard output. Ad esempio, il risultato del comando precedente può essere memorizzato nel file `result.xml` tramite il seguente comando:

```
$ ./EXPath -q /descendant::surname < turing.xml > result.xml
```

Oltre all'esecuzione di query, l'engine EXPath offre una serie di funzionalità aggiuntive come ad esempio quella specificata dal parametro `-m` che consente di ottenere informazioni sul numero di nodi e sullo spazio occupato in memoria dal *memory tree*. Ad esempio, per ottenere queste informazioni sul documento `turing.xml` è necessario utilizzare il seguente comando:

```
$ ./EXPath -m < turing.xml
Number of nodes in the memory-tree:  11
Bytes of the memory-tree:  445 bytes
```

Un'altra funzionalità interessante è quella relativa alla rappresentazione grafica del documento XML; in particolare tramite i parametri `-d` e `-dpp` è possibile generare una rappresentazione nel linguaggio Dot dell'albero XML: il parametro `-d` viene utilizzato per generare una rappresentazione base, mentre il parametro `-dpp` per una rappresentazione comprensiva dei valori *pre*/*post* per ogni nodo del *memory tree*.

Ad esempio per ottenere una rappresentazione del documento `turing.xml` nel linguaggio Dot, senza le informazioni *pre*/*post*, e memorizzare il risultato in un file denominato `turing.dot` è necessario eseguire il seguente comando:

```
$ ./EXPath -d < turing.xml > turing.dot
```

Il file `turing.dot` conterrà le seguenti informazioni:

```
digraph xml {
0 [label="person"];
0 -> 1;
1 [label="code",shape=box];
1 -> 2 [style=dotted];
2 [label="123",style=dotted];
0 -> 3 [style=dotted];
3 [label="text1",style=dotted];
0 -> 4;
4 [label="name"];
4 -> 5;
5 [label="age",shape=box];
5 -> 6 [style=dotted];
6 [label="42",style=dotted];
4 -> 7 [style=dotted];
7 [label="Alan",style=dotted];
0 -> 8 [style=dotted];
8 [label="text2",style=dotted];
0 -> 9;
9 [label="surname"];
9 -> 10 [style=dotted];
10 [label="Turing",style=dotted];
}
```

Successivamente utilizzando il programma *GraphViz* ed in particolare il seguente comando `dot`:

```
$ dot -Tps turing.dot -o turing.ps
```

si otterrà una rappresentazione in formato *PostScript* (.ps) dell'albero XML specificato nel file `turing.dot` generato dall'engine EXPath. In Figura 4.2 è riportato il file `turing.ps` generato tramite il software Dot ed in Figura 4.3 è riportata la rappresentazione dello stesso albero XML con l'aggiunta delle informazioni *pre*/*post* tramite il parametro `-dpp` dell'engine EXPath (si noti la simbologia utilizzata per differenziare la tipologia dei nodi: in particolare è stato utilizzato un cerchio per i

nodi *elemento*, un rettangolo per i nodi *attributo* ed un cerchio tratteggiato per i nodi di tipo *testo*).

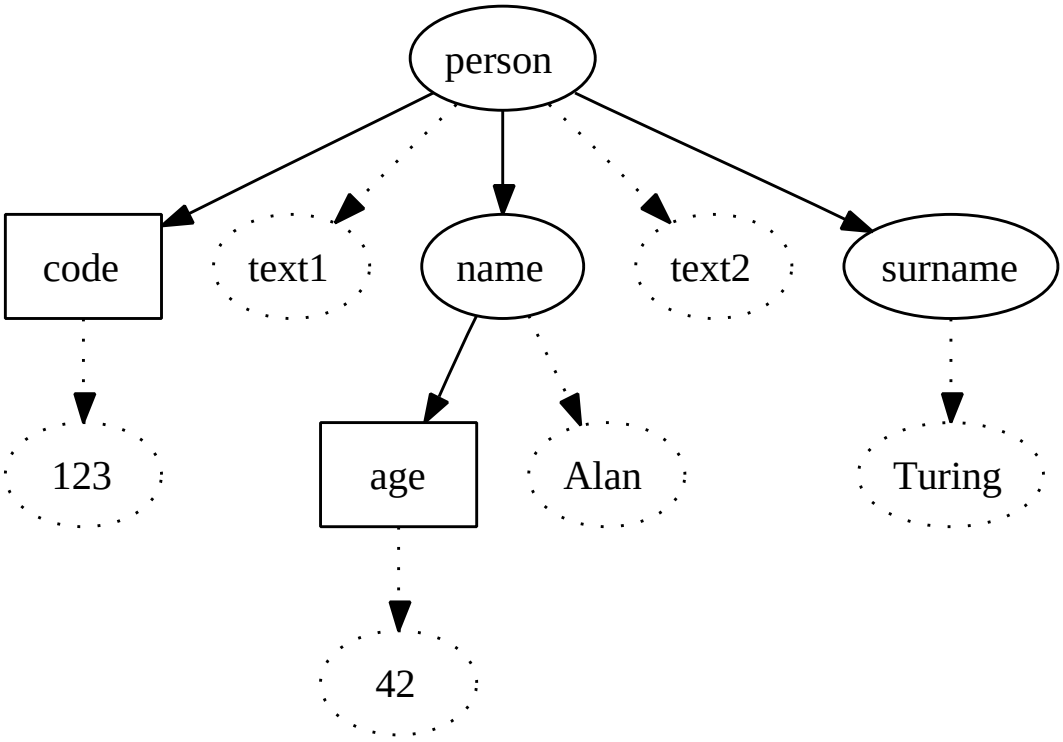

Figura 4.2: Rappresentazione ad albero del documento `turing.xml` generata dal programma Dot.

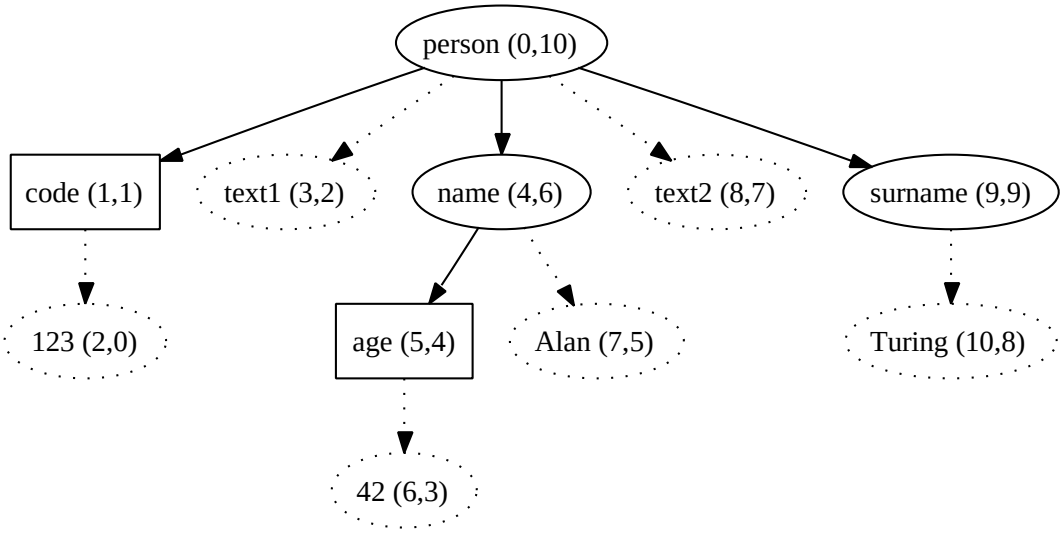

Figura 4.3: Rappresentazione ad albero del documento `turing.xml` con valori *pre/post* generata dal programma Dot.

Questa funzione dell'engine EXPath che consente di generare il diagramma in linguaggio Dot di un documento XML non era stata preventivata all'inizio dell'implementazione ed esula evidentemente dalle finalità della presente tesi di laurea. Ciò nonostante è una funzionalità aggiuntiva che si è rilevata particolarmente facile da implementare e di sicura utilità ed efficacia. Infatti, il fatto di avere una rappresentazione grafica dell'albero associato ad un file XML consente di osservare ed analizzare meglio i risultati di un'elaborazione di una query XPath. Ad esempio a partire da questa funzione dell'engine EXPath sarebbe interessante sviluppare un *front-end* grafico per la visualizzazione del risultato di interrogazioni XPath.

4.3 L'ambiente del test

Per poter valutare l'efficienza dell'engine EXPath è necessario effettuare alcuni test sui tempi di elaborazione di un insieme di query prestabilito su una famiglia di documenti XML di varie dimensioni. In questo modo è possibile variare la dimensione dell'input e verificare se i tempi di elaborazione sono effettivamente lineari in accordo con i risultati teorici ottenuti nel Capitolo 3 (si ricorda che nel nostro caso la dimensione dell'input è rappresentata dal prodotto $n \cdot k$, dove n indica la dimensione del documento XML e k la dimensione della query). Per poter effettuare questi test abbiamo utilizzato il progetto XMark [19] per la generazione di documenti XML di varie dimensioni ed il progetto XPathMark [20] per la scelta del campione di query da utilizzare. Oltre alla verifica della linearità dei tempi di risposta dell'engine EXPath si è effettuato un confronto con un engine famoso e molto efficiente, XMLTaskForce XPath, realizzato da Georg Gottlob, Christoph Koch e Reinhard Pichler [1, 14].

4.3.1 XMark, un benchmark per XML

XMark[7] è un progetto per la creazione di un benchmark XML nato dall'idea di alcuni ricercatori del CWI (Centrum Voor Wiskunde en Informatica) il centro di ricerca nazionale per la matematica e l'informatica dell'Olanda. Questo benchmark è modellato su una base dati di un'ipotetica asta on-line e consente di generare documenti XML di dimensioni variabili con relativo schema DTD associato. L'utilità di utilizzare un benchmark del genere consiste proprio nel fatto che i documenti XML che vengono generati non sono casuali ma corrispondono ad un modello nel mondo reale, in questo caso un sito d'aste on-line. Utilizzando documenti XML generati attraverso questo benchmark i test risultano più attendibili poichè più vicini alla complessità strutturale derivante dall'utilizzo di un caso d'uso reale. Oltre alla specifica della base dati dell'asta on-line, tramite lo schema DTD denominato `auction.dtd`, il benchmark XMark è costituito da un generatore di documenti XML denominato `xmlgen`.

Tramite la specifica di un numero denomiato *factor* si possono generare documenti XML con diverse dimensioni utilizzando il comando `xmlgen`. Ad esempio in

[7] http://www.xml-benchmark.org

ambiente Gnu/Linux con il seguente comando :

```
$ ./xmlgen.Linux -f 0.001 -o test.xml
```

viene generato un file denominato `test.xml` di dimensioni pari a 116 Kb. Al variare del parametro *factor* si possono generare documenti di qualsiasi dimensioni a partire da 26 Kb (f=0), ad esempio con f=1 si ottiene un documento XML di 116 Mb.

4.3.2 XPathMark, un benchmark per XPath

XPathMark[8] è un benchmark XPath per documenti XML generati con XMark ideato da Massimo Franceschet, ricercatore presso l'Informatics Institute dell'Università di Amsterdam e presso il Dipartimento di Scienze dell'Università degli Studi "G.D'Annunzio" di Pescara.

Questo benchmark consente di valutare la completezza funzionale, la correttezza, l'efficienza e la scalabilità di un engine XPath grazie all'utilizzo di 44 query XPath appositamente progettate sulla base dati dell'asta on-line del progetto XMark per gerantire l'utilizzo, in fase di testing, della maggior parte delle funzioni del linguaggio XPath 1.0. Così facendo i risultati sperimentali che questo benchmark è in grado di ottenere risultano essere completi sotto più profili.

Ad esempio le query sono suddivise nei seguenti gruppi: Axes, Node tests, Boolean operators, References proprio per differenziare la tipologia e la complessità delle interrogazioni. Ogni query ovviamente è stata progettata per fornire un risultato utile nella realtà: ad esempio la query Q22 restituisce gli oggetti in vendita nel Nord e nel Sud America, la query Q26 restituisce le aste aperte che una determinata persona stà osservando, e così via.

Le performance di un engine XPath vengono valutate da questo benchmark tramite l'utilizzo di alcuni indici che sintetizzano le caratteristiche dell'engine sotto più punti di vista.

[8] http://staff.science.uva.nl/~francesc/xpathmark

4.3.3 XMLTaskForce, un engine XPath

XMLTaskForce[9] è un engine XPath realizzato da Georg Gottlob, Christoph Koch e Reinhard Pichler in seguito alla pubblicazione dell'articolo "Efficient Algorithms for Processing XPath Queries" del 2002 [1].

Questo engine non supporta tutto il linguaggio XPath ma soltanto un sottoinsieme indicato più volte in questa tesi con il nome di Core XPath. XMLTaskForce è un engine efficiente con dei tempi di risposta lineari nelle dimensioni del documento XML e della query.

Per utilizzare questo engine è necessario specificare nella linea di comando la query Core XPath da elaborare ed il documento XML. Ad esempio in ambiente Gnu/Linux è necessario utilizzare il seguente comando:

```
$ ./xpath_i586_Linux query < file.xml
```

4.4 Risultati sperimentali

Di seguito sono riportati i risultati sperimentali ottenuti su un sistema Gnu/Linux 2.6.10 con processore AMD Sempron a 1.7 Ghz e 1 Gb di RAM. Per la generazione dei documenti XML è stato utilizzato il benchmark XMark con il generatore `xmlgen` ver. 0.92 utilizzando i seguenti valori di *factor*:

$$0.001, 0.002, 0.004, 0.008, 0.016, 0.032, 0.064, 0.128, 0.256, 0.512, 1$$

corrispondenti alle seguenti dimensioni in Mb:

$$0.116, 0.212, 0.468, 0.909, 1.891, 3.751, 7.303, 15.044, 29.887, 59.489, 116.517$$

Come engine XPath di confronto è stato utilizzato XMLTaskForce nell'ultima versione disponibile sul sito ufficiale del progetto www.xmltaskforce.com, ossia quella del 30-09-2004.

[9] http://www.xmltaskforce.com

Le query XPath utilizzate durante i test sono riportate in Figura 4.4. Le query da 1 a 5 corrispondono alle query Q1, Q2, Q3, Q4, Q6 del benchmark XPathMark mentre le altre sono state create per testare la velocità dell'engine EXPath sugli assi *descendant*, *ancestor-or-self*, *following*, *following-sibling*, *parent*, *child*, *preceding*, *preceding-sibling*.

Q1	`/child::site/child::regions/child::*/child::item`
Q2	`/child::site/child::closed_auctions/child::closed_auction/child::annotation/child::description/child::parlist/child::listitem/child::text/child::keyword`
Q3	`/descendant::keyword`
Q4	`/descendant-or-self::listitem/descendant-or-self::keyword`
Q5	`/descendant::keyword/ancestor::listitem`
Q6	`/descendant::keyword/ancestor-or-self::mail`
Q7	`/descendant::seller/following::*`
Q8	`/descendant::emailaddress/parent::person/child::name`
Q9	`/descendant::bidder/preceding::*`
Q10	`/descendant::seller/following-sibling::*/preceding-sibling::*`

Figura 4.4: Query XPath utilizzate per il test

Tutti i tempi di elaborazione riportati in questi test sono espressi in secondi e sono stati ottenuti tramite il comando *time* disponibile sui sistemi Gnu/Linux.

XMark factor	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
0.001	0.045	0.014	0.013	0.013	0.025	0.012	0.081	0.010	0.190	0.022
0.002	0.049	0.027	0.020	0.023	0.032	0.021	0.127	0.019	0.307	0.031
0.004	0.109	0.044	0.051	0.041	0.061	0.047	0.225	0.035	0.730	0.061
0.008	0.178	0.083	0.093	0.078	0.116	0.080	0.467	0.069	1.386	0.124
0.016	0.341	0.164	0.194	0.163	0.263	0.152	0.955	0.133	2.878	0.244
0.032	0.671	0.318	0.388	0.324	0.504	0.310	1.907	0.256	5.757	0.488
0.064	1.301	0.624	0.728	0.610	0.927	0.608	3.656	0.497	11.697	0.947
0.128	2.705	1.249	1.475	1.239	1.933	1.213	7.615	0.988	24.449	1.933
0.256	5.482	2.506	2.930	2.439	3.815	2.401	14.783	1.956	48.563	3.787
0.512	11.159	4.958	5.805	4.859	7.558	4.740	31.509	3.890	98.637	7.588
1.000	21.311	9.837	11.332	9.475	15.249	9.396	62.368	7.689	193.759	15.315

Figura 4.5: Tempi di elaborazione dell'engine XmlTaskForce (in secondi)

XMark factor	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
0.001	0.092	0.014	0.014	0.014	0.021	0.014	0.057	0.013	0.089	0.014
0.002	0.043	0.019	0.021	0.021	0.031	0.021	0.096	0.019	0.152	0.028
0.004	0.086	0.034	0.039	0.037	0.059	0.047	0.169	0.036	0.358	0.055
0.008	0.142	0.063	0.072	0.068	0.109	0.080	0.352	0.064	0.670	0.103
0.016	0.313	0.120	0.135	0.139	0.242	0.151	0.681	0.124	1.406	0.204
0.032	0.522	0.226	0.264	0.251	0.473	0.303	1.378	0.245	2.775	0.398
0.064	0.970	0.445	0.518	0.493	0.847	0.582	2.614	0.471	5.375	0.767
0.128	1.976	0.880	1.019	0.999	1.781	1.173	5.379	0.943	11.162	1.581
0.256	3.919	1.721	2.010	1.985	3.453	2.325	10.703	1.870	22.038	3.096
0.512	7.757	3.436	3.995	3.944	6.832	4.600	21.259	3.727	43.995	6.169
1.000	15.229	6.703	7.790	7.677	13.476	8.968	41.711	7.260	86.582	11.986

Figura 4.6: Tempi di elaborazione dell'engine EXPath (in secondi)

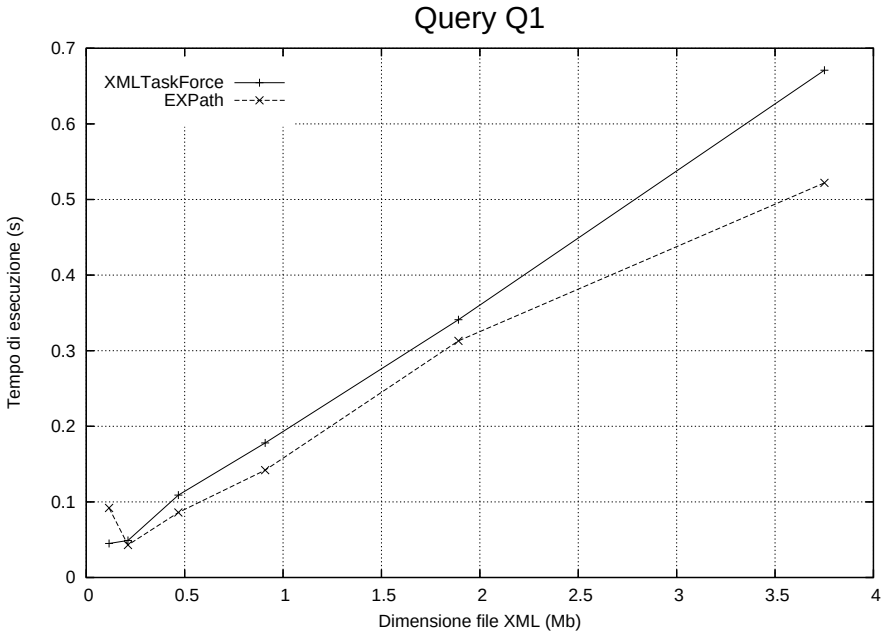

Figura 4.7: Query Q1 su documenti XML < 4 Mb

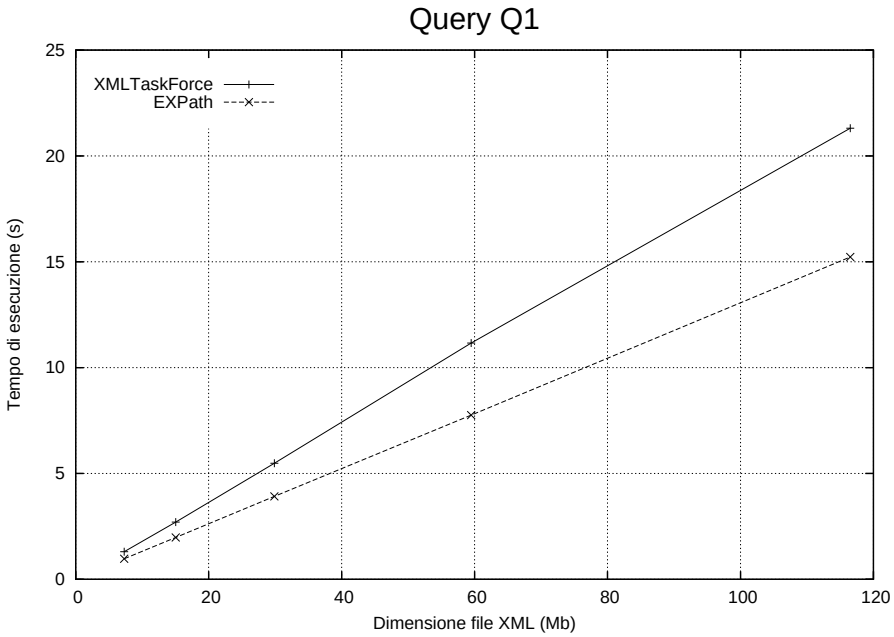

Figura 4.8: Query Q1 su documenti XML > 4 Mb

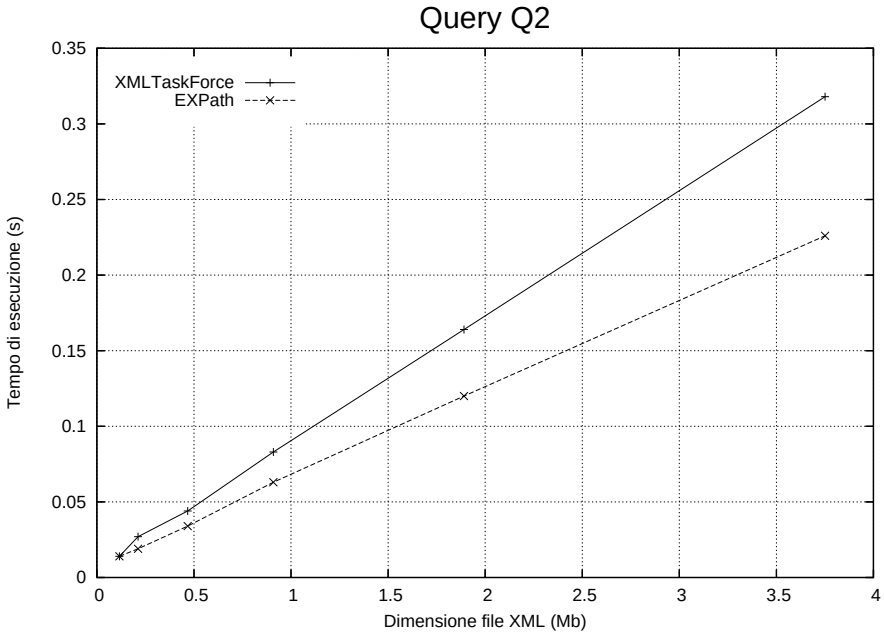

Figura 4.9: Query Q2 su documenti XML < 4 Mb

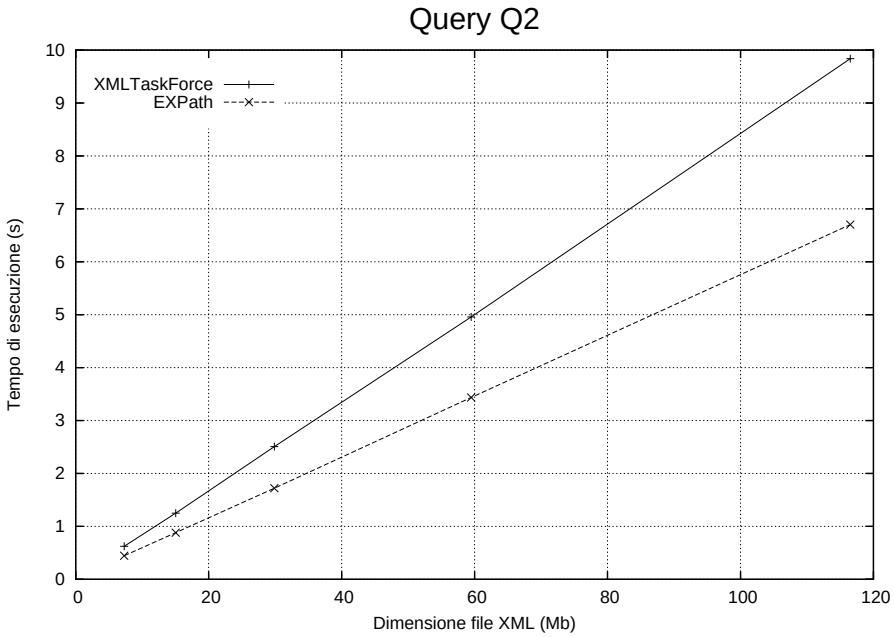

Figura 4.10: Query Q2 su documenti XML > 4 Mb

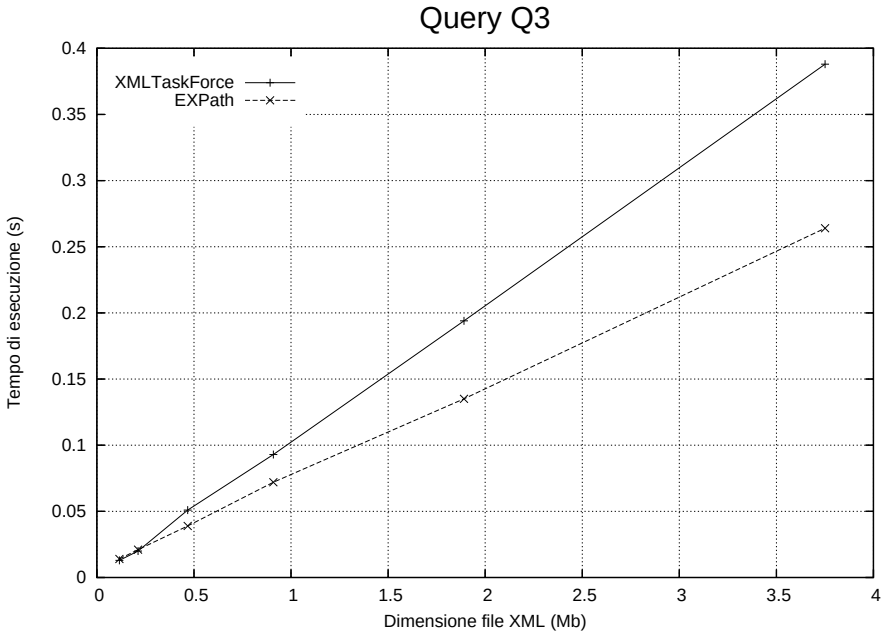

Figura 4.11: Query Q3 su documenti XML < 4 Mb

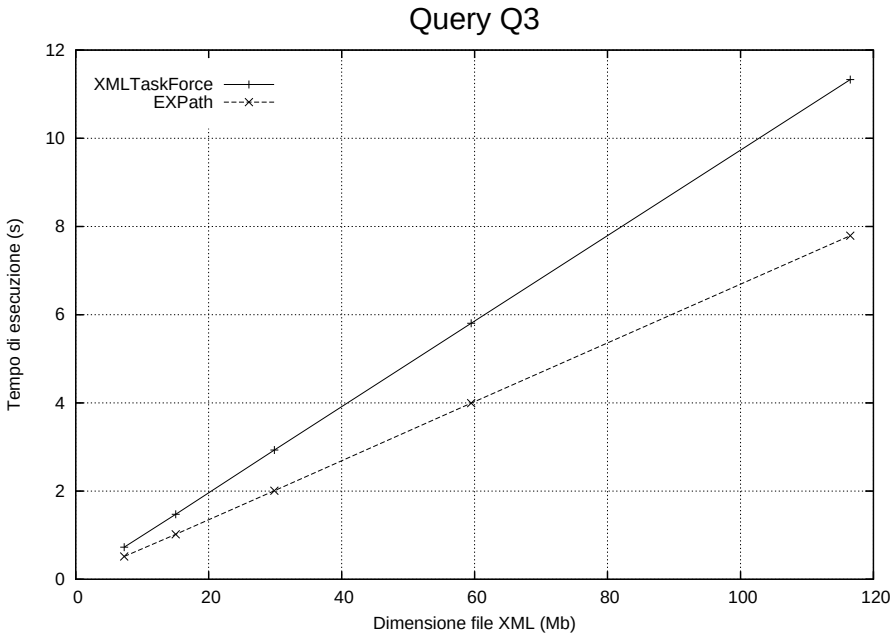

Figura 4.12: Query Q3 su documenti XML > 4 Mb

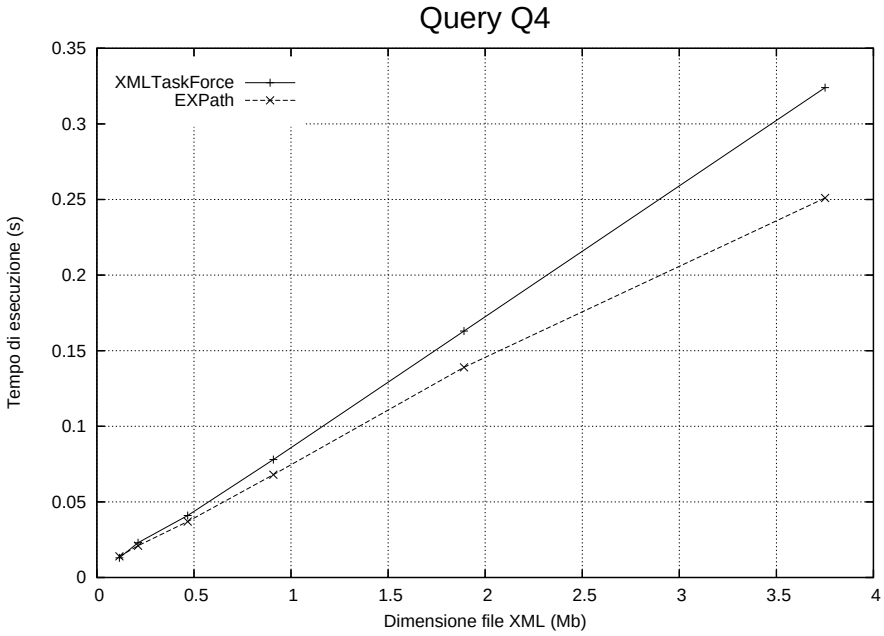

Figura 4.13: Query Q4 su documenti XML < 4 Mb

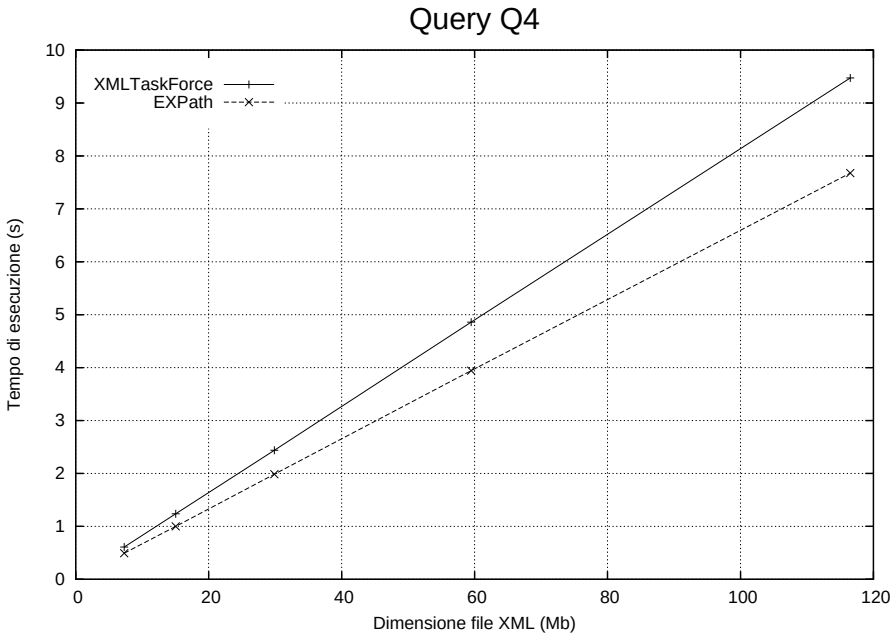

Figura 4.14: Query Q4 su documenti XML > 4 Mb

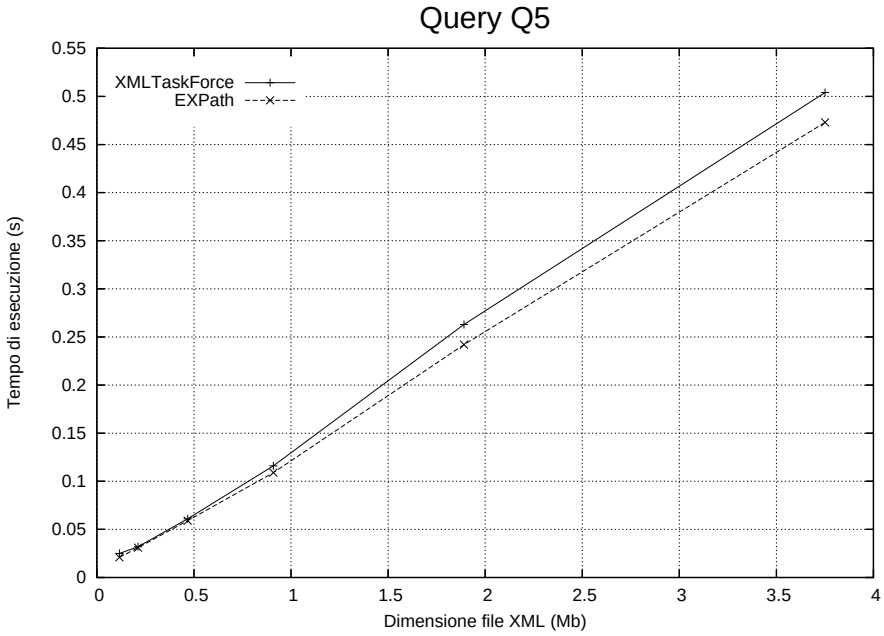

Figura 4.15: Query Q5 su documenti XML < 4 Mb

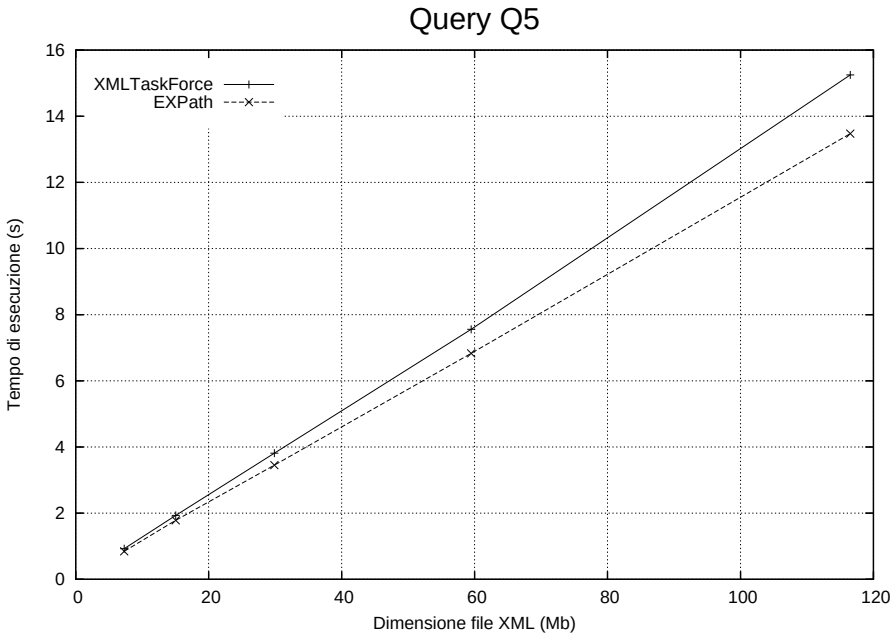

Figura 4.16: Query Q5 su documenti XML > 4 Mb

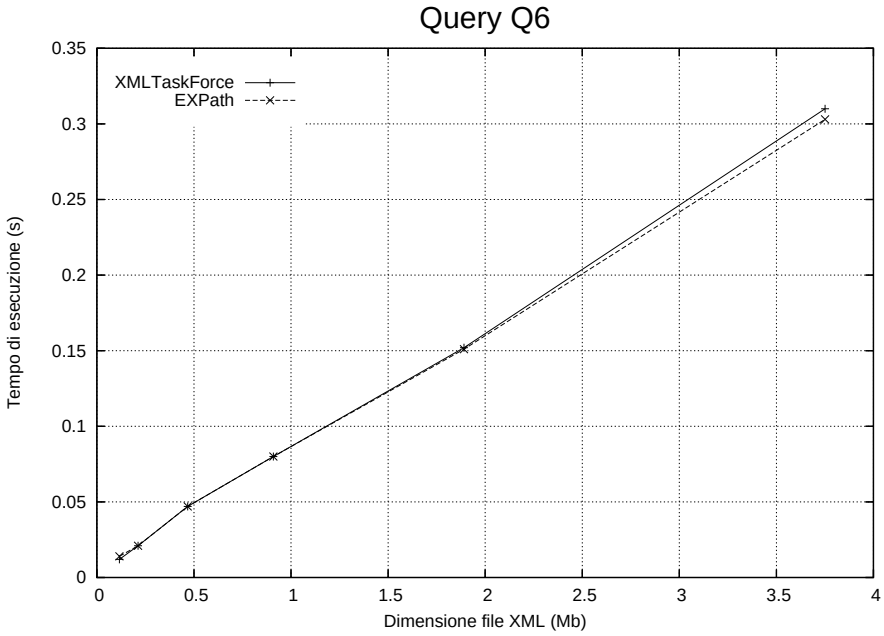

Figura 4.17: Query Q6 su documenti XML < 4 Mb

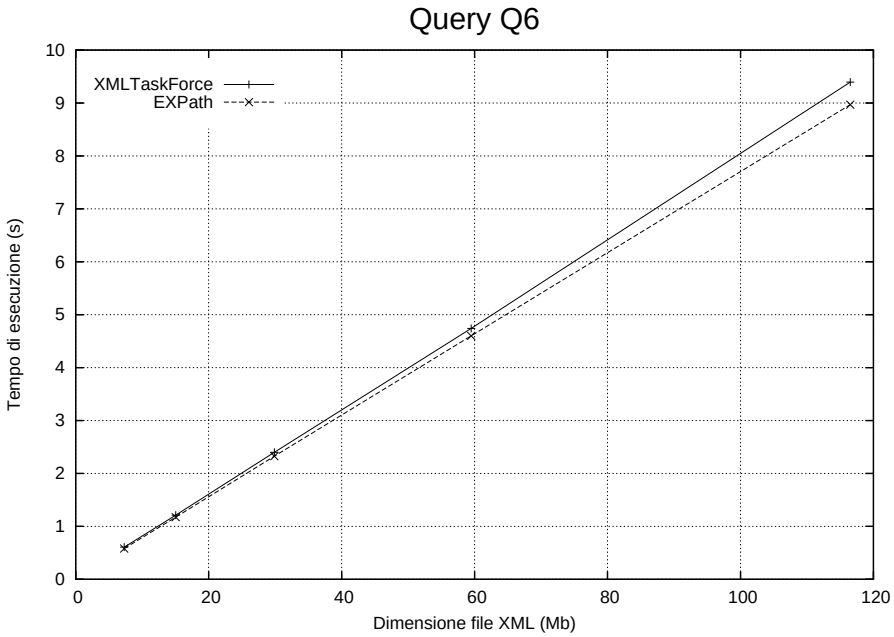

Figura 4.18: Query Q6 su documenti XML > 4 Mb

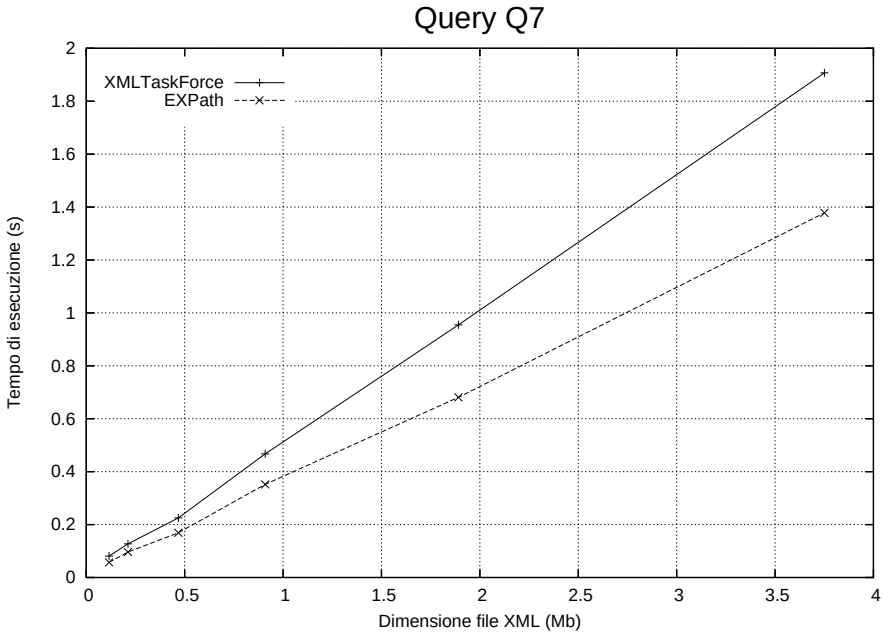

Figura 4.19: Query Q7 su documenti XML < 4 Mb

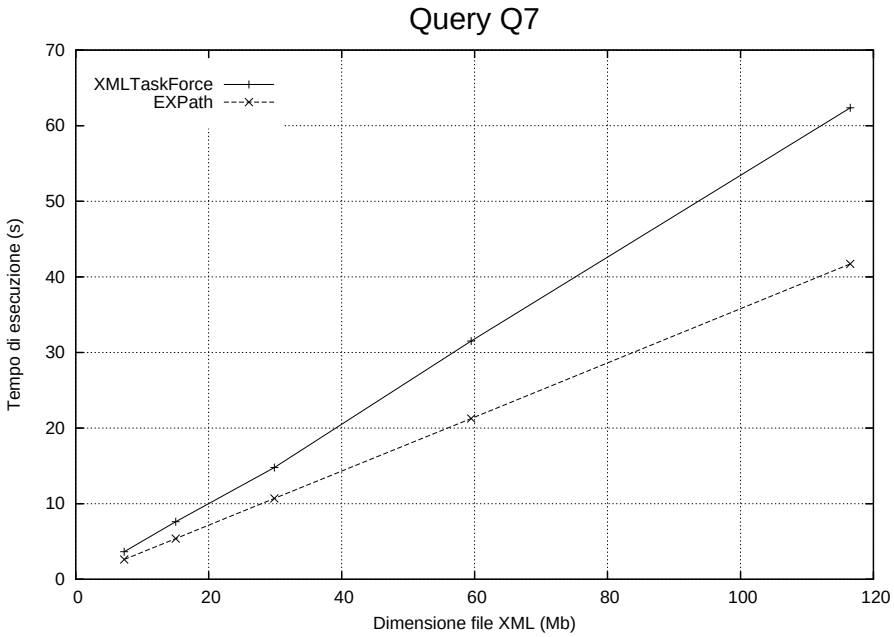

Figura 4.20: Query Q7 su documenti XML > 4 Mb

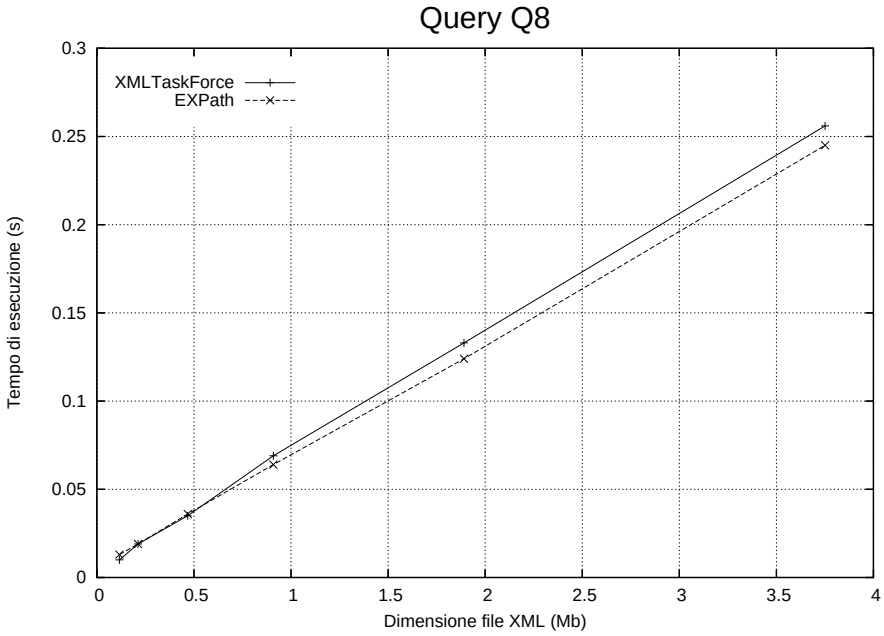

Figura 4.21: Query Q8 su documenti XML < 4 Mb

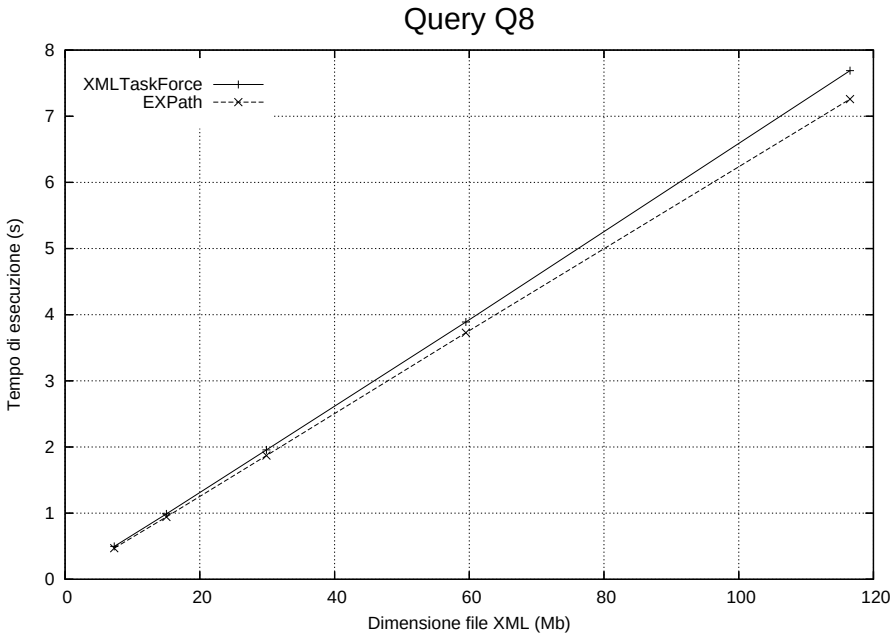

Figura 4.22: Query Q8 su documenti XML > 4 Mb

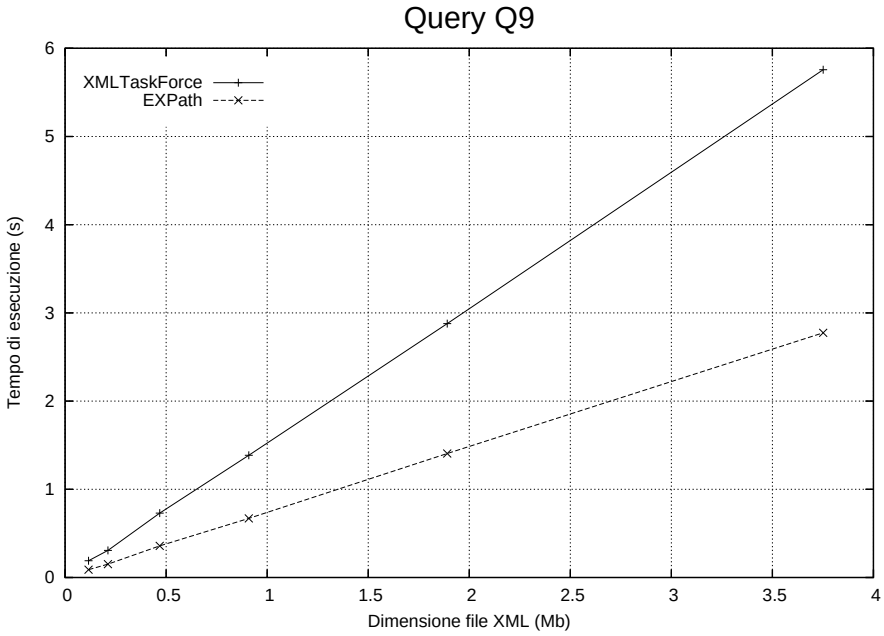

Figura 4.23: Query Q9 su documenti XML < 4 Mb

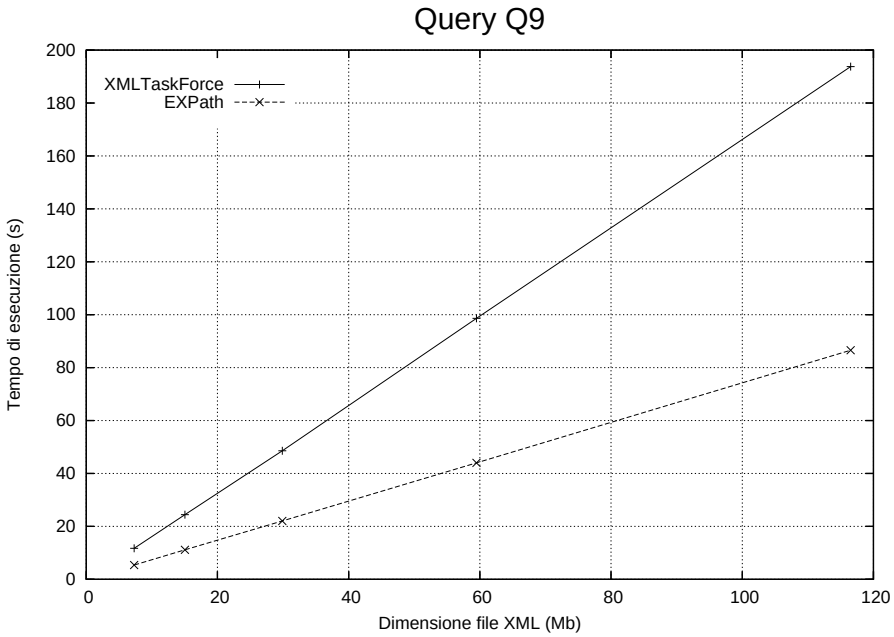

Figura 4.24: Query Q9 su documenti XML > 4 Mb

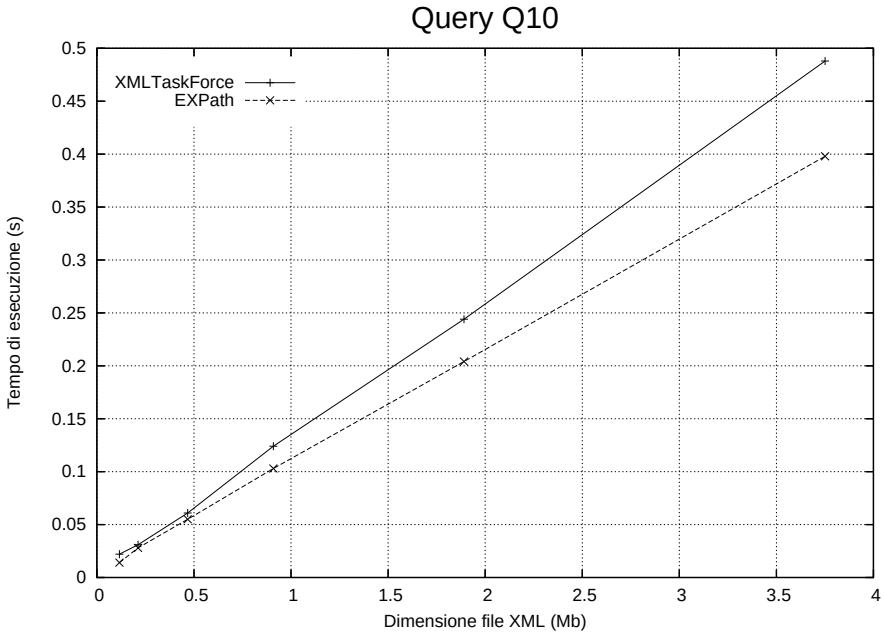

Figura 4.25: Query Q10 su documenti XML < 4 Mb

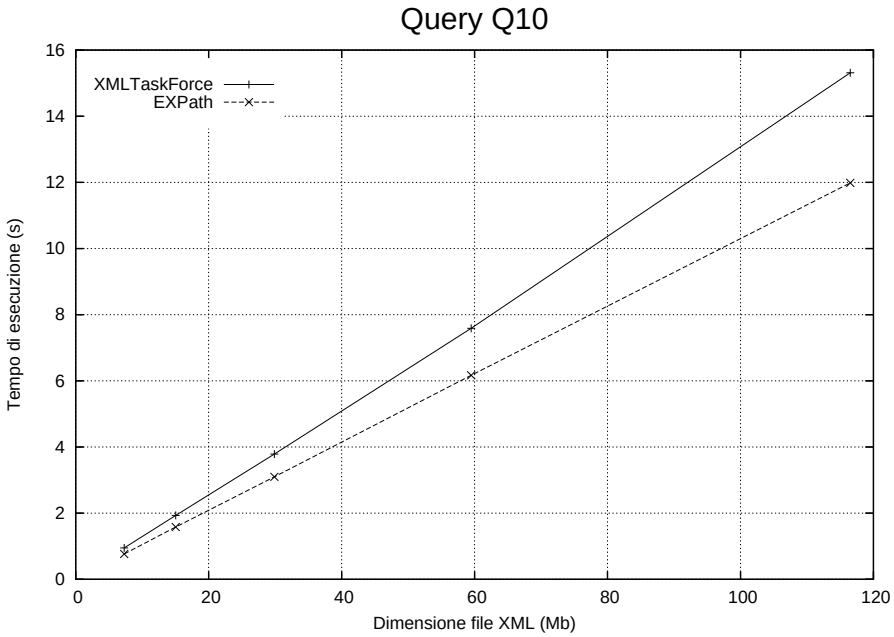

Figura 4.26: Query Q10 su documenti XML > 4 Mb

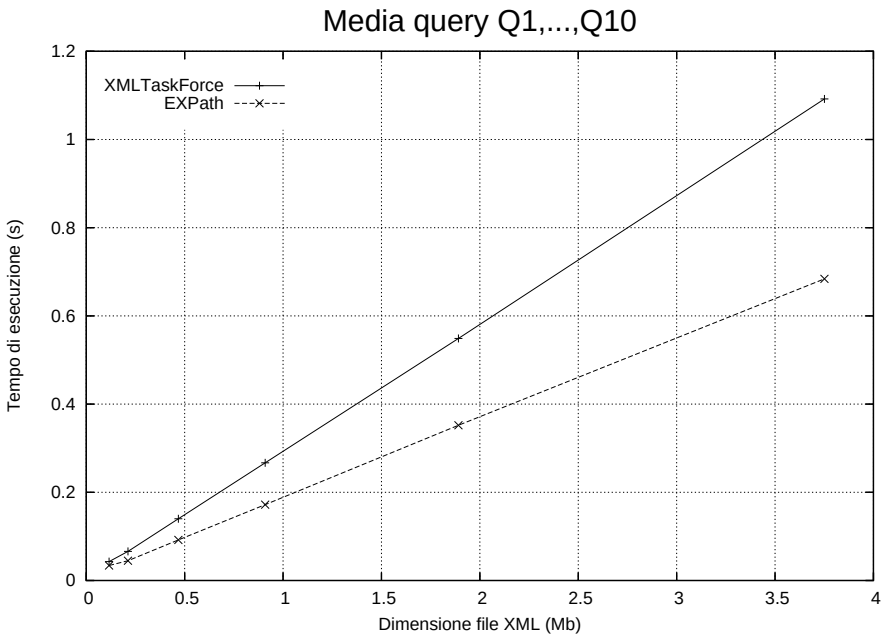

Figura 4.27: Media query Q1,...,Q10 su documenti XML < 4 Mb

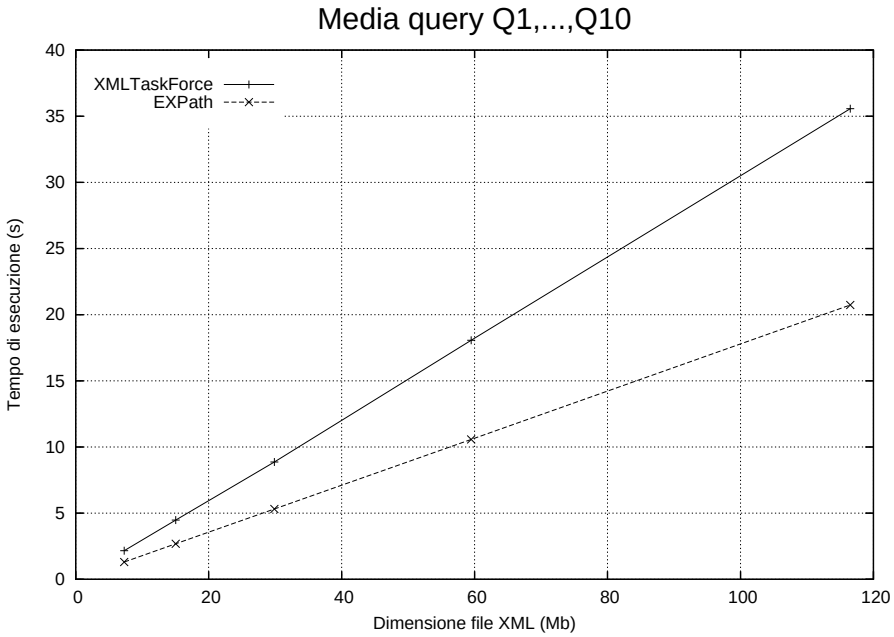

Figura 4.28: Media query Q1,...,Q10 su documenti XML > 4 Mb

4.5 Conclusioni

I test effettuati dimostrano che l'engine EXPath, in termini di velocità di esecuzione, ha un andamento lineare in funzione della dimensione del documento XML e risulta essere più efficiente dell'engine XMLTaskForce. In particolare osservando gli ultimi due grafici (Figura 4.27 e Figura 4.28) in cui sono riportate le medie dei tempi di elaborazione di tutte le *query*, si evince che l'engine EXPath è circa il doppio più efficiente di XMLTaskForce. In particolare calcolando il rapporto tra i tempi di elaborazione dei due engine XMLTaskForce/EXPath sulle medie di tutte le *query* Q1,...,Q10 otteniamo i valori riportati in Figura 4.29.

XMark factor	**Media XMLTaskForce (s)**	**Media EXPath (s)**	**XMLTaskForce/EXPath**
0.001	0.043	0.034	1.24
0.002	0.066	0.045	1.45
0.004	0.140	0.092	1.53
0.008	0.267	0.172	1.55
0.016	0.549	0.352	1.56
0.032	1.092	0.684	1.6
0.064	2.160	1.308	1.65
0.128	4.480	2.689	1.67
0.256	8.866	5.312	1.67
0.512	18.070	10.571	1.71
1.000	35.573	20.738	1.72

Figura 4.29: Tempi medi di elaborazione di tutte le query Q1,...,Q10

Calcolando la media sui valori dell'ultima colonna si ottiene un valore di 1.58: in media l'engine EXPath è più efficiente di XMLTaskForce, rispetto alle query utilizzate nel test, del 58%.

In particolare osservando l'elaborazione per ogni *query* risulta che Q1, Q2, Q3, Q7, Q9 sono molto più efficiente nell'engine EXPath rispetto a XMLTaskForce e che Q4, Q5 e Q10 risultano essere leggermente più efficienti nell'engine EXPath rispetto a XMLTaskForce; infine i risultati delle *query* Q6 e Q8 denotano più o meno la stessa efficienza degli engine.

Un altro fattore interessante da analizzare è la *scalabilità rispetto ai dati* (*data scalability*) ossia la capacità di elaborare interrogazioni in maniera efficiente all'aumentare della dimensione del documento XML. Consideriamo due documenti d_1 di

dimensione s_1 e d_2 di dimensione s_2 con $s_1 < s_2$ e una *query* q. Siano t_1 e t_2 i tempi di elaborazione della query q sui documenti d_1 e d_2 rispettivamente. Indichiamo con $v_1 = s_1/t_1$ la velocità di q su d_1 e $v_2 = s_2/t_2$ la velocità di q su d_2. La scalabilità della query q è definita come il rapporto tra v_1 e v_2 ossia:

$$\frac{v_1}{v_2} = \frac{t_2 \cdot s_1}{t_1 \cdot s_2}$$

Se tale fattore è minore di 1, quando $v_1 < v_2$, avremo una scalabilità di tipo *sub-lineare*. Se tale fattore è maggiore di 1, quando $v_1 > v_2$, allora avremo una scalabilità di tipo *super-lineare*. Nel caso in cui il fattore è uguale a 1 avremo una scalabilità di tipo *lineare*.

Un fattore di scalabilità *sub-lineare* indica che i tempi di elaborazione della query aumentano meno che linearmente all'aumentare delle dimensioni del documento XML mentre un fattore di scalabilità *super-lineare* indica che i tempi di elaborazione della query aumentano più che linearmente.

Nel nostro caso il fattore di scalabilità può essere calcolato considerando coppie di documenti XML consecutive del tipo $(f, 2f)$, dove f indica il *factor* XMark, e considerando i tempi medi di elaborazione delle query Q1,...Q10 riportati in Figura 4.29. Per ogni coppia di documenti XML $(f, 2f)$ il calcolo del rapporto v_1/v_2 risulta:

$$\frac{v_1}{v_2} = \frac{t_2 \cdot f}{t_1 \cdot 2f} = \frac{t_2}{2t_1}$$

Il calcolo di quest'operazione per ogni coppia di documenti XML consecutivi del tipo $(f, 2f)$ è riportato in Figura 4.30.

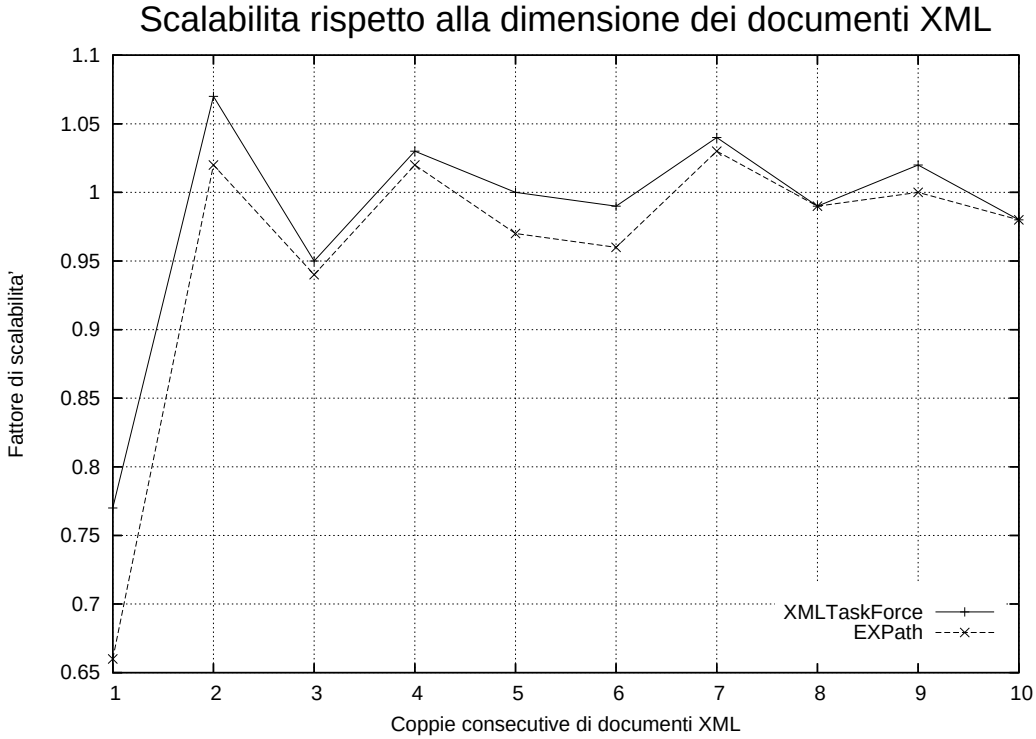

Figura 4.30: Scalabilità rispetto alla dimensione dei documenti XML

Come è possibile osservare dalla Figura 4.30 l'andamento del fattore di scalabilità di EXPath risulta essere per lo più *sub-lineare*, mentre il fattore di scalabilità di XMLTaskForce per lo più *super-lineare*. Ciò significa che l'engine EXPath risulta essere più scalabile dell'engine XMLTaskForce ossia che a parità di aumento delle dimensioni del documento XML l'engine EXPath elabora le query in maniera più efficiente rispetto a XMLTaskForce.

Concludendo, sia il fattore di *efficienza* in termini di velocità di elaborazione delle *query* che il fattore di *scalabilità* risultano essere superiori nell'engine EXPath rispetto all'engine XMLTaskForce. Questo è un risultato incoraggiante considerando anche che l'implementazione proposta in questa tesi di laurea non è completa e può essere ulteriormente ottimizzata.

Conclusioni

In questa tesi di laurea abbiamo esteso il linguaggio *Core XPath* introdotto da Gottlob, Koch e Pichler in [1] in un nuovo linguaggio, denominato *EXPath*, ed abbiamo dimostrato, nel Capitolo 3, come sia possibile risolvere interrogazioni in questo linguaggio in maniera efficiente su documenti XML con attributi e riferimenti.

L'efficienza dell'algoritmo proposto per la risoluzione di interrogazioni *EXPath* è di tipo lineare con complessità $O(k \cdot n)$, dove k è la dimensione dell'interrogazione ed n è la dimensione del documento XML. Questa complessità computazionale è la stessa ottenuta da Gottlob *et al.* in [1].

Oltre alla dimostrazione teorica del risultato di efficienza abbiamo proposto, nel Capitolo 4, un'implementazione dell'algoritmo, utilizzando il linguaggio C, di una parte fondamentale di *EXPath* verificandone sperimentalmente la complessità lineare su diversi documenti XML, generati con il benchmark *XMark* [19].

Come ultima fase del progetto di tesi abbiamo effettuato un confronto della nostra implementazione *EXPath* con un famoso engine, molto efficiente, *XMLTaskForce* realizzato da Gottlob *et al.* in [1].

I risultati ottenuti dal confronto con *XMLTaskForce* hanno evidenziato una superiorità in termini di *efficienza di elaborazione* ed in termini di *scalabilità* del nostro engine *EXPath*.

Questi risultati risultano essere particolarmente incoraggianti visto che l'implementazione, nel linguaggio C, proposta in questa tesi di laurea è da considerarsi un prototipo e che quindi sussistono ampi spazi di miglioramento.

Concludendo possiamo affermare che le tecniche del *pre/post* e del *flag numerico*, utilizzate per la realizzazione degli algoritmi presentati in questa tesi, sono risultate particolarmente efficaci tali da incoraggiare un proseguo degli studi in questo settore.

Bibliografia

[1] Georg Gottlob, Christopher Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *Very Large DataBases 2002 Conference*, Hong Kong, China, 2002.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, 2000.

[3] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Third Edition)*, 2004. URL: http://www.w3c.org/TR/2004/REC-xml-20040204/.

[4] World Wide Web Consortium. *HTML 4.01 Specification*, 1999.
URL: http://www.w3.org/TR/html4/.

[5] Dan Suciu. On database theory and XML. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, Vol. 30(Num. 3):pagg. 39–45, September 2001.

[6] Ronald Bourret. Xml and databases. Technical report, 2004.
URL: http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[7] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts with Oracle CD*. McGraw-Hill Science/Engineering/Math, 4 edition, 2001.

[8] University of Michigan. Timber: a native XML Database system, 2004.
URL: http://www.eecs.umich.edu/db/timber/.

[9] Wolfgang Meier. eXist: Open Source Native XML Database, 2000.
URL: http://exist.sourceforge.net/.

[10] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, 1999. URL: http://www.w3c.org/TR/xpath.

[11] Steven Holzner. *XPath, navigating XML with XPath 1.0 and 2.0.* Sams Publishing, 2004.

[12] World Wide Web Consortium. *Document Object Model (DOM) Level 1 Specification*, 1998. URL: http://www.w3.org/TR/REC-DOM-Level-1/.

[13] Philip Wadler. Two semantics for XPath. Technical report, Bell Labs, 2000. URL: http://homepages.inf.ed.ac.uk/wadler/topics/xml.html.

[14] Georg Gottlob, Christopher Koch, and Reinhard Pichler. XPath query evaluation: Improving time and space efficiency. In *IEEE International Conference on Data Engineering (ICDE)*, 2003.

[15] Jan Hidders and Philippe Michiels. Efficient XPath Axis Evaluation for DOM Data Structures. *PLAN-X, Venice (Italy)*, 2004.

[16] Torsten Grust. Accelerating XPath location steps. In *SIGMOD Conference*, 2002.

[17] Torsten Grust, Jan Hidders, Philippe Michiels, Roel Vercammen, and Maurice van Keulen. Supporting positional predicates in efficient XPath axis evaluation for DOM data structures. Technical Report TR2004-05, University of Antwerp and University of Twente and University of Konstanz, 2004.

[18] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Very Large DataBases 2003 Conference*, Berlin, Germany, 2003.

[19] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[20] M. Franceschet. XPathMark – an XPath benchmark for XMark. Technical Report PP-2004-04, ILLC, University of Amsterdam, The Netherlands, 2005. URL: http://dare.uva.nl/document/12761.

[21] L. Afanasiev, M. Franceschet, M. J. Marx, and Rijke Rijke. *CTL model checking for processing simple XPath queries*. IEEE Computer Society Press, 2004.

[22] Elliote Rusty Harold and W.Scott Means. *XML in a nutshell*. O'Reilly, 2 edition, 2001.

[23] Paolo Pialorsi. *XML il nuovo linguaggio del Web*. Mondadori Informatica, 2002.

[24] Paolo Pialorsi. *Programmare con XML*. Mondadori Informatica, 2004.

Appendice - Sorgenti C

EXPath.c

```
/*****************************************************************
 * EXPath ver. 0.1
 *
 * Read an XML document and process the EXPath query.
 *
 * Copyright 2005, Enrico Zimuel (enrico@zimuel.it)
 *
 * License: GNU General Public License (GPL)
 *****************************************************************/

#include <stdio.h>
#include <limits.h>
#include "memorytree.h"

struct element_list {                   /* an element on the list */
   struct node   *c;
   struct element_list   *previous;
   struct element_list   *next;
};

typedef  struct element_list element_list;

struct list {                   /* a list */
   struct element_list  *first;
   struct element_list  *last;
};

typedef  struct list list;

list ListNode;
struct node *root;  /* This is the root node of the xml memory-tree */
int CONT;       /* This is the global var used by numeric flag technic */


/* BEGIN - List routine */
```

```
boolean EmptyList(const list *lst)
{
   return ((boolean) (lst -> first == NULL));
}


struct list  *NewList()
{
   struct list *lst;

   lst = malloc(sizeof(list));

   lst -> first = NULL;
   lst -> last = NULL;

   return lst;
}

struct node *DelFirst(struct list *lst)
{
   struct node *d= NULL;
   struct element_list *p;

   if (lst -> first != NULL) {
     p = lst -> first;
     d = lst -> first -> c;
     lst -> first = lst -> first -> next;
     free (p);
     if (lst -> first != NULL) {
       lst -> first -> previous = NULL;
     }
   }

   return d;
}

struct node *DelLast(struct list *lst)
{
   struct node *d = NULL;
   struct element_list *p;

   if (lst -> last != NULL) {
     p = lst -> last;
     d = lst -> last -> c;
     lst -> last = lst -> last -> previous;
     free (p);
     if (lst -> last != NULL) {
       lst -> last -> next = NULL;
```

```
     }
   }

   return d;
}

void AddAfter(struct list *lst, struct node *n)
{
  element_list *p;

  p= malloc(sizeof(element_list));
  p -> c = n;
  p -> previous = NULL;
  p -> next = NULL;

  if (lst -> first == NULL) {
    lst -> first = p;
    lst -> last = p;
  } else {
    p -> previous = lst -> last;
    lst -> last -> next = p;
    lst -> last = p;
  }

}

void AddListAfter(struct list *lst1, struct list *lst2)
{

  if ((lst1 -> first == NULL) && (lst2 -> first != NULL)) {
    lst1 -> first = lst2 -> first;
    lst1 -> last = lst2 -> last;
  } else if (lst2 -> first != NULL) {
    lst1 -> last -> next = lst2 -> first;
    lst2 -> first -> previous = lst1 -> last;
    lst1 -> last = lst2 -> last;
  }

}

void AddBefore(struct list *lst, struct node *n)
{
  element_list *p;

  p= malloc(sizeof(element_list));
  p -> c = n;
  p -> previous = NULL;
  p -> next = NULL;

  if (lst -> first == NULL) {
```

```
    lst -> first = p;
    lst -> last = p;
  } else {
    p -> next = lst -> first;
    lst -> first -> previous = p;
    lst -> first = p;
  }

}

void AddListBefore(struct list *lst1, struct list *lst2)
{

  if ((lst1 -> first == NULL) && (lst2 -> first != NULL)) {
    lst1 -> first = lst2 -> first;
    lst1 -> last = lst2 -> last;
  } else if (lst2 -> first != NULL) {
    lst1 -> first -> previous = lst2 -> last;
    lst2 -> last -> next = lst1 -> first;
    lst1 -> first = lst2 -> first;
  }

}

struct node *First(struct list *lst)
{
   if (lst->first == NULL) {
     return NULL;
   } else {
     return lst -> first -> c;
   }
}

struct node *Last(struct list *lst)
{
   if (lst->last == NULL) {
     return NULL;
   } else {
     return lst -> last -> c;
   }
}

/* END - List routine */


/* BEGIN - Tree routine */

struct node *FirstChild(struct node *n, tnode t)
{
  struct node *x = NULL;
```

```
  if (n != NULL) {

    x = n -> child;

    if (t != all) {

      if (t == attribute) {

        if ((x != NULL) && ((x -> type == element) || (x -> type == text)))
  x = NULL;

      } else {

        while ((x != NULL) && (x -> type != t))
  x = x -> right_sibling;
      }

    }

  }

  return x;

}

struct node *RightSibling(struct node *n, tnode t)
{
  struct node *x = NULL;

  if (n != NULL) {

    x = n -> right_sibling;

    if (t != all) {

      if (t == attribute) {

        if ((x != NULL) && ((x -> type == element) || (x -> type == text)))
  x = NULL;

      } else {

        while ((x != NULL) && (x -> type != t))
  x = x -> right_sibling;

      }
    }
  }
```

```
  return x;

}

struct node *LeftSibling(struct node *n, tnode t)
{
  struct node *x = NULL;

  if (n != NULL) {

    x = n -> left_sibling;

    if (t != all) {

      if ((t == element) || (t == text)) {

        while ((x != NULL) && (x -> type != t) && ((x -> type != attribute) || (x -> type != ID) ||
         (x -> type != IDREF) || (x -> type != IDREFS)))
          x = x -> left_sibling;

      }
    }
  }

  return x;
}

struct node *ParentNode(struct node *n, tnode t)
{
  struct node *x = NULL;

  if (n != NULL) {

    x = n -> parent;

   if (t != all) {
     while ((x != NULL) && (x -> type != t))
       x = x -> parent;
   }

  }

  return x;

}

/* END - Tree routine */


/* BEGIN - Axes algorithms */
```

```
void AllText (struct node *n, char *value) {

  struct node *n1;

  n1= FirstChild(n,all);

  while (n1 != NULL) {

    if (n1 -> type == text) {
      value= strcat(value, n1 -> tag);
    } else {
      if (n1 -> type == element) {
        AllText (n1,value);
      }
    }

    n1 = RightSibling (n1,all);
  }

}


char *S(struct node *n) {

  char *value;
  struct node *n1;

  if (n -> type == text) {
    value = malloc(strlen(n -> tag)+1);
    value = (char *) n -> tag;
  } else if (n -> type == element) {
    AllText (n,value);
  } else {
    n1 = FirstChild(n,text);
    value = malloc(strlen(n1 -> tag)+1);
    value = (char *) n1 -> tag;
  }

  return value;
}


struct list *Equal(struct list *C,char *s) {

  struct list *L;
  struct node *n;

  L= NewList();
```

```
  while (!EmptyList(C)) {
    n = DelFirst(C);
    if (strcmp(S(n),s)==0) {
      AddAfter (L,n);
    }
  }

  return L;
}


struct list *Self(struct list *C, char *a) {

  struct list *L;
  struct node *n;

  L= NewList();

  while (!EmptyList(C)) {
    n = DelFirst(C);
    if ((n -> type == element) && ((strcmp(n->tag,a)==0) || (strcmp(a,"*")==0))) {
      AddAfter (L,n);
    }
  }

  return L;
}

struct list *AllChildren(struct node *n, char *a) {

  struct list *L;
  struct node *n1;

  L= NewList();
  n1= FirstChild(n,element);

  while (n1 != NULL) {
    if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
      AddAfter (L,n1);
    }
    n1= RightSibling(n1,element);
  }

  return L;
}

struct list *Child(struct list *C, char *a) {

  struct list *L;
  struct list *S;
```

```
  struct node *n;

  L= NewList();
  S= NewList();

  while (!EmptyList(C)) {
    n = First(C);
    if (EmptyList(S)) {
      AddListBefore (S,AllChildren(n,a));
      n = DelFirst(C);
    } else if (First(S) -> pre <= n -> pre) {
      AddAfter (L,DelFirst(S));
    } else {
      AddListBefore (S,AllChildren(n,a));
      n = DelFirst(C);
    }
  }

  if (!EmptyList(S)) {
    AddListAfter (L,S);
  }

  return L;
}

struct list *Parent(struct list *C, char *a) {

  struct list *L;
  struct node *n1;
  struct node *n;

  L= NewList();

  while (!EmptyList(C)) {
    n = DelFirst(C);
    n1 = ParentNode(n,element);
    if ((n1 -> flag < CONT) && ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0))) {
      AddAfter(L,n1);
      n1 -> flag = CONT;
    }
  }

  return L;
}

struct list *SelfAttribute(struct list *C, char *a) {

  struct list *L;
  struct node *n;
```

```
  L= NewList();

  while (!EmptyList(C)) {
    n = DelFirst(C);
    if (((n -> type == attribute) || (n -> type == ID) || (n -> type == IDREF) ||
       (n -> type == IDREFS)) && ((strcmp(n->tag,a)==0) || (strcmp(a,"*")==0))) {
      AddAfter(L,n);
    }
  }

  return L;
}

struct list *Attribute(struct list *C, char *a) {

  struct list *L;
  struct node *n;
  struct node *n1;

  L= NewList();

  while (!EmptyList(C)) {
    n = DelFirst(C);
    n1 = FirstChild(n,attribute);

    while (n1 != NULL) {
      if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
        AddAfter (L,n1);
      }
      n1 = RightSibling(n1,attribute);
    }

  }

  return L;
}

struct list *ParentAttribute(struct list *C, char *a) {

  struct list *L;
  struct node *n;
  struct node *n1;

  L= NewList();

  while (!EmptyList(C)) {

    n = DelFirst(C);

    if ((n -> type == attribute) || (n -> type == ID) || (n -> type == IDREF) || (n -> type == IDREFS)) {
```

```
      n1 = ParentNode(n,element);

      if ((n1 -> flag < CONT) && ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0))) {
        AddAfter (L,n1);
        n1 -> flag = CONT;
      }
    }
  }

  return L;
}

void AddDescendant(struct list *L, struct node *n, char *a) {

  struct node *n1;

  n1 = FirstChild(n,element);

  while ((n1 != NULL) && (n1 -> flag < CONT)) {
    if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
      AddAfter (L,n1);
    }
    n1 -> flag = CONT;
    AddDescendant (L,n1,a);
    n1 = RightSibling (n1,element);
  }

}

struct list *Descendant(struct list *C, char *a, int self) {

  struct list *L;
  struct node *n;

  L= NewList();

  while (!EmptyList(C)) {
    n = DelFirst(C);
    if ((self == 1) && ((strcmp(n->tag,a)==0) || (strcmp(a,"*")==0))) {
      AddAfter (L,n);
      n -> flag = CONT;
    }
    AddDescendant (L,n,a);
  }

  return L;
}

struct list *Ancestor(struct list *C, char *a, int self) {
```

```
  struct list *L;
  struct list *S;
  struct node *n;
  struct node *n1;

  L= NewList();

  while (!EmptyList(C)) {
    n = DelFirst(C);
    S= NewList();

    if (self == 1){
      n1 = n;
    } else {
      n1 = ParentNode(n,element);
    }

    while ((n1 != NULL) && (n1 -> flag < CONT)) {
      if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
        AddBefore (S,n1);
      }
      n1 -> flag = CONT;
      n1 = ParentNode(n1,element);
    }
    AddListAfter (L,S);
  }

  return L;
}

void AddAllDescendant(struct list *L, struct node *n, char *a) {

  struct node *n1;

  n1 = FirstChild(n,element);

  while (n1 != NULL) {
    if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
      AddAfter (L,n1);
    }
    n1 -> flag = CONT;
    AddAllDescendant (L,n1,a);
    n1 = RightSibling (n1,element);
  }

}

struct list *Following(struct list *C, char *a) {
```

```
  struct list *L;
  struct node *n;
  struct node *n1;

  L= NewList();

  if (!EmptyList(C)) {
    n = DelFirst(C);

    while (!EmptyList(C) && (First(C) -> post < n -> post)) {
      n = DelFirst(C);
    }

    while (n != NULL) {
      n1= RightSibling (n,element);
      while (n1 != NULL) {
        if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
          AddAfter (L,n1);
        }
        AddAllDescendant (L,n1,a);
        n1 = RightSibling (n1,element);
      }
      n = ParentNode(n,element);
    }

  }

  return L;
}

struct list *FollowingSibling(struct list *C, char *a) {

  struct list *L;
  struct list *H;
  struct list *S;
  struct node *n;
  struct node *n1;

  L= NewList();
  H= NewList();

  while (!EmptyList(C)) {

    S = NewList();
    n = DelFirst(C);
    n1 = RightSibling(n,element);

    while ((n1 != NULL) && (n1 -> flag < CONT)) {
      if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
        if ((!EmptyList(C)) && (First(C) -> post < n1 -> post)) {
```

```
          AddAfter (S,n1);
        } else {
          while ((!EmptyList(H)) && (First(H) -> pre < n1 -> pre)) {
            AddAfter(L,DelFirst(H));
          }
          AddAfter(L,n1);
        }
      }
      n1 -> flag = CONT;
      n1 = RightSibling (n1,element);
    }
    AddListBefore(H,S);
  }
  AddListAfter(L,H);

  return L;
}

struct list *Preceding(struct list *C, char *a) {

  struct list *L;
  struct list *S;
  struct node *n;
  struct node *n1;

  L= NewList();
  n= Last(C);

  while (n != NULL) {

    n1 = LeftSibling(n,element);
    S= NewList();

    while (n1 != NULL) {
      if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
        AddAfter (S,n1);
      }
      n1 -> flag = CONT;
      AddAllDescendant (S,n1,a);
      n1 = LeftSibling (n1,element);
    }

    AddListBefore (L,S);
    n = ParentNode(n,element);
  }

  return L;
}

struct list *PrecedingSibling(struct list *C, char *a) {
```

```
  struct list *L;
  struct list *H;
  struct list *S;
  struct node *n;
  struct node *n1;

  L= NewList();
  H= NewList();

  while (!EmptyList(C)) {

    S = NewList();
    n = DelLast(C);
    n1 = LeftSibling(n,element);

    while ((n1 != NULL) && (n1 -> flag < CONT)) {
      if ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0)) {
        if ((!EmptyList(C)) && (Last(C) -> pre > n1 -> pre)) {
          AddBefore (S,n1);
        } else {
          while ((!EmptyList(H)) && (Last(H) -> pre > n1 -> pre)) {
            AddBefore(L,DelLast(H));
          }
          AddBefore(L,n1);
        }
      }
      n1 -> flag = CONT;
      n1 = LeftSibling (n1,element);
    }
    AddListAfter(H,S);
  }
  AddListBefore(L,H);

  return L;
}

struct node *NextNode (struct node *n) {

  struct node *n1;

  n1 = RightSibling (n,element);
  while ((n1 == NULL) && (n != NULL) && (n -> flag < CONT)) {
    n = ParentNode (n,element);
    n -> flag = CONT;
    n1 = RightSibling(n,element);
  }

  return n1;
}
```

```
struct list *Next(struct list *C, char *a, int sibling) {

  struct list *L;
  struct list *S;
  struct node *n;
  struct node *n1;

  L= NewList();
  S= NewList();

  while (!EmptyList(C)) {

    n = DelFirst(C);

    if (sibling == 1) {
      n1 = RightSibling(n,element);
    } else {
      n1 = NextNode(n);
    }

    if ((n1 != NULL) && (n1 -> flag < CONT) && ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0))) {
      if ((!EmptyList(C)) && (First(C) -> post < n1 -> post)) {
        AddBefore (S,n1);
      } else {
        while ((!EmptyList(S)) && (First(S) -> pre < n1 -> pre)) {
          AddAfter (L,DelFirst(S));
        }
        AddAfter (L,n1);
      }
      n1 -> flag = CONT;
    }
  }

  AddListAfter(L,S);

  return L;
}

struct node *PreviousNode (struct node *n) {

  struct node *n1;

  n1 = LeftSibling (n,element);
  while ((n1 == NULL) && (n != NULL) && (n -> flag < CONT)) {
    n = ParentNode (n,element);
    n -> flag = CONT;
    n1 = LeftSibling(n,element);
  }
```

```
  return n1;
}

struct list *Previous(struct list *C, char *a, int sibling) {

  struct list *L;
  struct list *S;
  struct node *n;
  struct node *n1;

  L= NewList();
  S= NewList();

  while (!EmptyList(C)) {

    n = DelLast(C);

    if (sibling == 1) {
      n1 = LeftSibling(n,element);
    } else {
      n1 = PreviousNode(n);
    }

    if ((n1 != NULL) && (n1 -> flag < CONT) && ((strcmp(n1->tag,a)==0) || (strcmp(a,"*")==0))) {
      if ((!EmptyList(C)) && (Last(C) -> pre > n1 -> pre)) {
        AddBefore (S,n1);
      } else {
        while ((!EmptyList(S)) && (Last(S) -> pre > n1 -> pre)) {
          AddAfter (L,DelLast(S));
        }
        AddBefore (L,n1);
      }
      n1 -> flag = CONT;
    }
  }

  AddListBefore(L,S);

  return L;
}

/* END - Axes algorithms */

void ResetTreeFlag(struct node *n) {

  if (n!=NULL) {
    n -> flag = 0;
    n= n->child;
    while (n!=NULL) {
      ResetTreeFlag(n);
```

```
      n = n->right_sibling;
    }
  }

}


struct list *Evaluate(char *axis, char *a, struct list *C) {

  if (strcmp(axis,"self")==0) C = Self(C,a);
  else if (strcmp(axis,"child")==0) C = Child(C,a);
  else if (strcmp(axis,"parent")==0) C = Parent(C,a);
  else if (strcmp(axis,"self-attribute")==0) C = SelfAttribute(C,a);
  else if (strcmp(axis,"attribute")==0) C = Attribute(C,a);
  else if (strcmp(axis,"parent-attribute")==0) C = ParentAttribute(C,a);
  else if (strcmp(axis,"descendant")==0) C = Descendant(C,a,0);
  else if (strcmp(axis,"descendant-or-self")==0) C = Descendant(C,a,1);
  else if (strcmp(axis,"ancestor")==0) C = Ancestor(C,a,0);
  else if (strcmp(axis,"ancestor-or-self")==0) C = Ancestor(C,a,1);
  else if (strcmp(axis,"following")==0) C = Following(C,a);
  else if (strcmp(axis,"following-sibling")==0) C = FollowingSibling(C,a);
  else if (strcmp(axis,"preceding")==0) C = Preceding(C,a);
  else if (strcmp(axis,"preceding-sibling")==0) C = PrecedingSibling(C,a);
  else if (strcmp(axis,"next")==0) C = Next(C,a,0);
  else if (strcmp(axis,"next-sibling")==0) C = Next(C,a,1);
  else if (strcmp(axis,"previous")==0) C = Previous(C,a,0);
  else if (strcmp(axis,"previous-sibling")==0) C = Previous(C,a,1);
  else {
    printf ("Error: the '%s' axe speficied is not valid!\n", axis);
    exit(1);
  }

  CONT++;
  if (CONT == INT_MAX) {
    ResetTreeFlag(root);
    CONT = 1;

  }

  return C;
}


// Elaborate the EXPath query q for all elements of the list C
struct list *EXPath (struct list *C, char *q) {

  char *q1;

  if (!EmptyList(C)) {
    if (*q == '/') {
```

```
      C = EXPath (C,q+1);
    } else if ((q1=strchr(q,'/'))!= NULL) {

      *q1 = '\0';
      C = EXPath(EXPath(C,q),q1+1);

    } else if ((q1=strchr(q,':'))!= NULL) {

      *q1 = '\0';
      C = Evaluate(q,q1+2,C);

    } else {
      printf("The query specified isn't a valid EXPath query!\n");
      exit(1);
    }
  }

  return C;
}


void PrintXmlNode(struct node *n) {

  struct node *n1;

  if ((n!=NULL) && (n->type == element)) {

    printf ("<%s",n->tag);

    n1= FirstChild(n,attribute);
    while (n1 != NULL) {
      printf(" %s=\"%s\"", n1->tag, n1->child->tag);
      n1 = RightSibling(n1,attribute);
    }
    printf (">");

    n1= FirstChild(n,all);
    while (n1 != NULL) {
      if (n1 -> type == text) {
        printf ("%s",n1->tag);
      } else if (n1 -> type == element) {
        PrintXmlNode(n1);
      }
      n1= RightSibling(n1,all);
    }
    printf ("</%s>\n", n->tag);
  }

}
```

```
void PreVisit(struct node *n) {

  if (n!=NULL) {
    printf ("%s (%d,%d)\n", n->tag, n->pre, n->post);
    n= n->child;
    while (n!=NULL) {
      PreVisit(n);
      n = n->right_sibling;
    }
  }

}


void DotDiagram(struct node *n, int pp) {

  struct node *x;

  if (n!=NULL) {

    if (pp==1) {
      if (n->type == element) {
        printf ("%d [label=\"%s (%d,%d)\"];\n", n->pre, n->tag, n->pre, n->post);
      } else if (n->type == text) {
        printf ("%d [label=\"%s (%d,%d)\",style=dotted];\n", n->pre, n->tag, n->pre, n->post);
      } else {
        printf ("%d [label=\"%s (%d,%d)\",shape=box];\n", n->pre, n->tag, n->pre, n->post);
      }
    } else {
      if (n->type == element) {
        printf ("%d [label=\"%s\"];\n", n->pre, n->tag);
      } else if (n->type == text) {
        printf ("%d [label=\"%s\",style=dotted];\n", n->pre, n->tag);
      } else {
        printf ("%d [label=\"%s\",shape=box];\n", n->pre, n->tag);
      }
    }

    x= n->child;
    while (x!=NULL) {
      if (x->type == text) {
        printf ("%d -> %d [style=dotted];\n", n->pre, x->pre);
      } else {
        printf ("%d -> %d;\n", n->pre, x->pre);
      }
      DotDiagram(x,pp);
      x= x->right_sibling;
    }
```

```
  }

}


void PrintDotDiagram(struct node *n,int pp) {

  printf ("digraph xml {\n");
  DotDiagram(n,pp);
  printf ("}\n");

}


int
main(int argc, char *argv[])
{
  int n_node;
  int byte_node;

  if (argc < 2) {
    printf("EXPath ver. 0.1\n");
    printf("by Enrico Zimuel (enrico@zimuel.it)\n");
    printf("Usage: ./EXPath [-q query] [-m] [-d] [-dpp] < file.xml\n");
    printf("-q query   elaborate the EXPath query\n");
    printf("-m         print info about the memory-tree of the xml file\n");
    printf("-d         print the memory-tree in the graphviz diagram format\n");
    printf("-dpp       print the memory-tree in the graphviz diagram format with pre/post value\n");
    printf("For more information about graphviz: http://www.graphviz.org\n");
    exit (1);
  }

  // Initialize the global var used by numeric flag
  CONT= 1;

  // Parse the XML file and create the memory-tree with root node
  root= XmlTree(root, &n_node, &byte_node);

  if (strcmp(argv[1],"-q")==0) {

    struct list *C;
    struct node *n;
    char *query;

    n = malloc(sizeof(struct node));
    n -> type = element;
    n -> tag = "";
    n -> child = root;
    n -> parent = NULL;
    n -> right_sibling = NULL;
```

```
    n -> left_sibling = NULL;


    C= NewList();
    AddAfter(C,n);

    query= malloc(strlen(argv[2])+1);
    query= strcpy(query,argv[2]);

    // Elaborate the query specified
    C= EXPath(C,query);

    n = DelFirst(C);
    while (n != NULL) {
      PrintXmlNode(n);
      n = DelFirst(C);
    }

  } else
  if (strcmp(argv[1],"-m")==0) {
     printf ("Number of nodes in the memory-tree: %d\n",n_node);
     printf ("Bytes of the memory-tree: %d bytes\n",byte_node);
  } else if (strcmp(argv[1],"-d")==0) {
     PrintDotDiagram(root,0);
  } else if (strcmp(argv[1],"-dpp")==0) {
     PrintDotDiagram(root,1);
  }

  return 0;
}
```

memorytree.h

```
/*
Library for create XML memory-tree.
*/

#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE     8192

#define BUFFSIZE     8192

#define   EMPTY      0
#define   FULL       10000

typedef enum {false, true}   boolean;
typedef enum {all, element, text, attribute, ID, IDREF, IDREFS} tnode;

struct node {
tnode type;
const char *tag;
long pre,post;
int flag;
struct node *parent,*child,*right_sibling,*left_sibling;
};

struct elem {                   /* an element on the stack */
   struct node   *d;
   struct elem   *next;
};

typedef   struct elem   elem;

struct stack {
   int    cnt;                  /* a count of the elements */
   elem   *top;                 /* ptr to the top element */
};

typedef   struct stack   stack;

char Buff[BUFFSIZE];
int num_node;
int byte_node;
stack up,left;
struct node *root_mt;


// Stack functions
boolean   empty(const stack *stk);
```

```
boolean   full(const stack *stk);
void      initialize(stack *stk);
void      push(struct node *d, stack *stk);
struct node *pop(stack *stk);
struct node *top(stack *stk);

// XML Tree functions
struct node *AddNode(const char *s,int num, tnode t);
static void XMLCALL
start(void *data, const char *el, const char **attr);
static void XMLCALL
end(void *data, const char *el);
static void XMLCALL
charhndl(void *data, const char *s, int len);
struct node *XmlTree(struct node *root_mt2, int *num_node2, int *byte_node2);
```

memorytree.c

```
/* The basic stack routines. */

#include "memorytree.h"

// Stack routine (begin) ---------------------------------------

boolean stack_empty(const stack *stk)
{
   return ((boolean) (stk -> cnt == EMPTY));
}

boolean stack_full(const stack *stk)
{
   return ((boolean) (stk -> cnt == FULL));
}

void stack_initialize(stack *stk)
{
   stk -> cnt = 0;
   stk -> top = NULL;
}

void push(struct node *d, stack *stk)
{
   elem   *p;

   p = malloc(sizeof(elem));
   p -> d = d;
   p -> next = stk -> top;
   stk -> top = p;
   stk -> cnt++;
}

struct node *pop(stack *stk)
{
   struct node  *d;
   elem   *p;

   d = stk -> top -> d;
   p = stk -> top;
   stk -> top = stk -> top -> next;
   stk -> cnt--;
   free(p);
   return d;
}

struct node *top(stack *stk)
{
```

```
   return (stk -> top -> d);
}

// Stack routine (end) ------------------------------------


struct node *AddNode(const char *s,int num, tnode t)
{
  struct node *n;

  n = malloc(sizeof(struct node));
  n -> type= t;
  n -> tag = malloc(strlen(s)+1);
  n -> tag = strcpy ((char *) n -> tag,s);
  n -> pre= num;
  n -> post=0;
  n -> flag=0;
  n -> parent = NULL;
  n -> child = NULL;
  n -> right_sibling = NULL;
  n -> left_sibling = NULL;

  byte_node+= sizeof(struct node)+strlen(s);
  return n;
}

static void XMLCALL
start(void *data, const char *el, const char **attr)
{
  int i;
  struct node *n;
  struct node *x;
  struct node *m;
  struct node *y;

  // Add an element node
  n = AddNode(el,num_node,element);

  if (root_mt==NULL) {
    root_mt = n;
  }

  // Modify the node links

  if (!stack_empty(&up)) {
    x= top(&up);
  } else {
    x= NULL;
  }
```

```
n -> parent = x;

if ((x != NULL) && (x -> child == NULL)) {
  x -> child = n;
}

push(n,&up);

if (!stack_empty(&left)) {
  x= pop(&left);
} else {
  x= NULL;
}

n -> left_sibling = x;

if ((x != NULL) && (x -> right_sibling == NULL)) {
  x -> right_sibling = n;
}

m = NULL;
x = NULL;
num_node++;

// Insert the attribute nodes of the element
for (i = 0; attr[i]; i += 2) {

  x= AddNode(attr[i],num_node,attribute);

  x -> parent = n;

  if (n -> child == NULL) {
    n -> child = x;
  }

  x -> left_sibling = m;

  if (m != NULL) {
    m -> right_sibling = x;
  }

  m= x;
  num_node++;

  y= AddNode(attr[i+1],num_node,text);

  y -> parent = x;
  x -> child = y;
  num_node++;
```

```
  }

  if (x!=NULL) {
    push(x,&left);
  }

}

static void XMLCALL
end(void *data, const char *el)
{
  struct node *n;

  while (!stack_empty(&left)) {
    n= pop(&left);
  }

  if (!stack_empty(&up)) {
    n= pop(&up);
    push (n,&left);
  }

}

static void XMLCALL
charhndl(void *data, const char *s, int len)
{
  struct node *n;
  struct node *x;
  int i;
  char *txt;

  i=0;

  // Delete the first space char
  while (i<len && isspace(*s)) {
    s++;
    i++;
  }

  if (i<len) {

    txt= malloc(len+1-i);
    txt= strncpy(txt,s,len-i);

    n = AddNode(txt,num_node,text);

    if (!stack_empty(&up)) {
      x = top(&up);
    } else {
```

```
      x = NULL;
    }

    n -> parent = x;

    if ((x!=NULL) && (x -> child == NULL)) {
      x -> child = n;
    }

    if (!stack_empty(&left)) {
      x = pop(&left);
    } else {
      x = NULL;
    }

    n -> left_sibling = x;

    if ((x != NULL) && (x -> right_sibling == NULL)) {
      x -> right_sibling = n;
    }

    push (n,&left);
    num_node++;
  }

}

void CalcPost(struct node *n) {

  if (n!=NULL) {
    CalcPost(n->child);
    n->post= num_node++;
    CalcPost(n->right_sibling);
  }

}

struct node *XmlTree(struct node *root_mt2, int *num_node2, int *byte_node2) {

  // Inizialize the stacks
  stack_initialize (&up);
  stack_initialize (&left);
  num_node=0;
  byte_node=0;

  XML_Parser p = XML_ParserCreate(NULL);
  if (! p) {
    fprintf(stderr, "Couldn't allocate memory for parser\n");
    exit(-1);
  }
```

```
  XML_SetParamEntityParsing(p,XML_PARAM_ENTITY_PARSING_ALWAYS);

  XML_SetElementHandler(p, start, end);
  XML_SetCharacterDataHandler(p, charhndl);

  for (;;) {
    int done;
    int len;

    len = fread(Buff, 1, BUFFSIZE, stdin);
    if (ferror(stdin)) {
      fprintf(stderr, "Read error\n");
      exit(-1);
    }
    done = feof(stdin);

    if (XML_Parse(p, Buff, len, done) == XML_STATUS_ERROR) {
      fprintf(stderr, "Parse error at line %d:\n%s\n",
              XML_GetCurrentLineNumber(p),
              XML_ErrorString(XML_GetErrorCode(p)));
      exit(-1);
    }

    if (done)
      break;
  }

  *num_node2= num_node;
  *byte_node2= byte_node;

  // Insert post value in the memory-tree
  num_node=0;
  CalcPost(root_mt);

  root_mt2= root_mt;

  return root_mt2;
}
```