A Flexible Labeling Scheme for Efficient Structural Join over XML Graph*

Kuen-Fang Jea, Chien-Ping Chou, Shih-Ying Chen
Department of Computer Science, National Chung-Hsing University
Taichung 402, Taiwan, R.O.C.
{kfjea, phd9402, sychen}@cs.nchu.edu.tw

ABSTRACT

The most important operation of querying XML documents is finding occurrences of structural relationships between tagged elements. Performing such structural joins needs to encode nodes in XML documents. Once each node is labeled by the interval-based labeling scheme, structural relationships between nodes can be determined easily in constant time. However this paper shows that traditional approaches work only for tree-structured data and cannot be applied to XML documents being modeled as arbitrary graphs.

The main contribution of this paper is the solution of matching structural relationships in graph-structured XML data, under a new interval-based labeling scheme. Based on the idea of e-nodes, each XML node can be labeled (with an optional E-List) appropriately by our G_encoding algorithm for solving the multiple-inheritance and cyclic-path problems. We have formally analyzed the properties of the new labeling scheme, and developed an efficient structural join algorithm SJG using the scheme.

1: Introduction

The most important operation of querying XML data is finding occurrences of structural relationships between tagged elements, such as ancestor-descendant and parent-children relationships. For example, an XQuery expression: //*bank*[*branch_city*= 'Brooklyn']//*account* returns all accounts of every branch bank located in the 'Brooklyn' city. There exist three structural relationships of (ancestor, descendant) in the example, including (*bank*, *branch_city*), (*branch_city*, 'Brooklyn') and (*bank*, *account*). Several solutions for matching such structural relationships in XML or semi-structured documents were proposed. Zhang *et al.* [11] presented a multi-predicate merge-join (MPMGJN) algorithm, which outperforms the traditional merge-join and index nested-loop join algorithms. Al-Khalifa *et al.* [2] proposed two families of structural join algorithms, namely Tree-Merge-Anc/Desc and Stack-Tree-Anc/Desc. Recently papers by Chien *et al.* [4] or Kim *et al.* [9] proposed even more efficient structural join algorithms by skipping unnecessary node scan in different ways. For example, several index-based solutions for structural join algorithms using XB-tree [3], B+-tree [4] or XR-tree [8] were proposed.

Performing structural joins in [2][4][9][10][11] needs to encode nodes in XML documents. The position of an XML element is usually represented as a 3-tuple (*DocId*, *StartPos*: *EndPos*, *LevelNum*) [2][4][9][11] where *DocId* is the document identifier, *LevelNum* is the nesting depth, and *StartPos* and *EndPos* are generated by counting word numbers from the beginning of the document to the start and to the end of the element respectively. A depth-first-search (DFS) [5] algorithm is employed in encoding (labeling), which explores all the elements in the top-down and left-to-right manner. Structural relationships between elements can be determined easily by comparing their intervals between *StartPos* and *EndPos*. A tree node *u* is an ancestor of a tree node *v* if and only if *u.DocId* = *v.DocId*, *u.StartPos* < *v.StartPos*, and *u.EndPos* > *v.EndPos*; a parent-child relationship also requires that *u.LevelNum* = *v.LevelNum* – 1. For example, Figure 1(a) is an XML document with six elements, and Figure 1(b) illustrates the result of DFS labeling. The DFS algorithm visits nodes *A*, *B*, *C*, *D*, *E* and *F* in turn. Node *A* is labeled as (1, 1: 12, 1), node *B* is labeled as (1, 2: 5, 2), and so on.

The IDREF and IDREFS features in XML documents make multiple-inheritance possible, as several paths may exist from one node to another. Element sharing is specified using element IDs and IDREFS, and XML data can then specify nested and cyclic structures, such as trees, directed-acyclic graphs, and arbitrary graphs. Some well-known XML query languages, like XML-QL [6] and Lorel [1], support matching path expressions for graph-structured XML data. However, the proposed structural join solutions [2][4][9][11] are limited to the tree-structured XML data. They cannot manage arbitrary graphs with multiple-inheritance, or even with cyclic paths. As shown in Figure 1(b), both nodes *B* (1, 2: 5, 2) and *D* (1, 6: 7, 2) are ancestors of node *C* (1, 3: 4, 3), but the interval of node *D*, i.e., (6: 7), does not contain that of node *C*, i.e., (3: 4). Here we can see that the existing interval-based labeling schemes fail because they only work for trees. To overcome this limitation, we propose in this paper a node-labeling scheme for a rooted directed cyclic graph allowing multiple-inheritance and cyclic paths. Further, we propose a structural join

* This research is supported in part by NSC in Taiwan, R.O.C. under Grant No. NSC-95-2221-E-005-048.

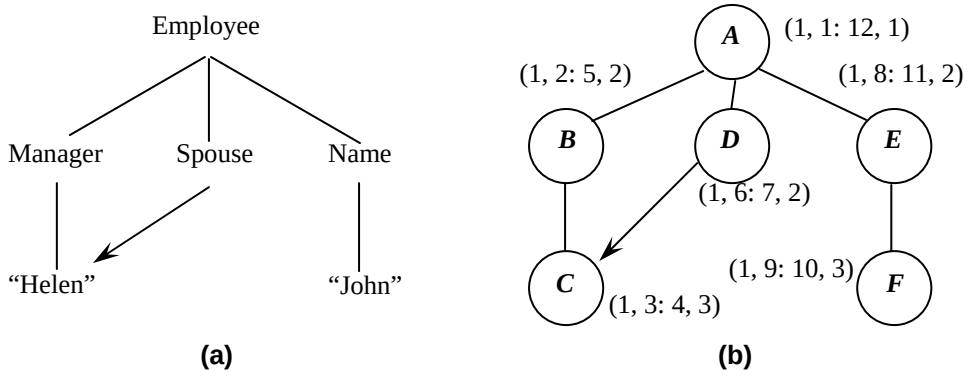

Figure 1. An XML document and DFS labeling.

algorithm based on this node-labeling scheme to match structural relationships in graph-structured XML data.

The rest of the paper is organized as follows. Section 2 describes the new labeling scheme, followed by a structural join algorithm using this new scheme in Section 3. Finally Section 4 concludes this study and discusses its future work.

2: Node Labeling Scheme

Our labeling scheme is based on the idea of *e-nodes* for labeling nodes over XML data. A labeling algorithm derived from DFS [5] is also developed to support structural joins for XML graphs.

2.1: E-Lists and E-nodes

Our labeling scheme starts by labeling nodes in a graph with an extended DFS algorithm. During DFS each node (element/string) in XML documents is labeled as (*DocId*, *StartPos*: *EndPos*, *LevelNum*), whose format is the same as that in [2][4][9][11]. We will omit *DocId* below if no confusion occurs.

Definition 1. *Let graph G = (V, E) where V is a set of XML document nodes and E is a set of edges between nodes in V. Edges (u, v)∈E where u, v∈V are classified during DFS labeling.* ***Tree edges*** *are those edges (u, v) if vertex v is first visited by exploring edge (u, v).* ***Back edges*** *are those edges (u, v) connecting a vertex u to an ancestor v.* ***Forward edges*** *are those non-tree edges (u, v) connecting a vertex u to a descendant v.* ***Cross edges*** *are any other edges. We use the "Labeled DFS Graph" to indicate a graph after DFS labeling.*

Both *forward edges* and *cross edges* point to nodes that have been explored completely. Without loss of generality, *cross edges* are regarded as *forward edges* in a *Labeled DFS Graph* because our labeling scheme treats them in the same way.

For simplicity, we use *b/f edge* to indicate *back/forward edge* and use (*S*: *E*, *L*) to denote (*StartPos*: *EndPos*, *LevelNum*). *Interval (S: E)* denotes the range from *StartPos* to *EndPos*, and $(S_u: E_u, L_u)$ denotes *(u.S: u.E, u.L)* for certain node *u*.

In our scheme, we construct an *e-node(v)* for each *back/forward edge (u, v)* and store it into a list associated with node *u*, namely *E-List(u)*. Note that our scheme does not distinguish *forward* and *cross edges*. *E-node(v)* is derived from node *v* and labeled as *(S: E, L)* = $(S_v: E_v, L_v - 1)$. Due to the transitivity rule, whenever an *e-node* is stored into *E-List(u)*, it should also be saved in *E-List(m)* for each node *m* on the path from *u* to the root comprising only *tree edges*. Conversely, whenever an *e-node(v)* is in *E-List(u)*, all *e-nodes* in *E-List(v)* are all in *E-List(u)*. Copying *e-nodes* from *E-List(v)* to *E-List(u)* implies the existence of a path consisting of multiple *b/f edges*. *E-node's LevelNum* will decrease 1 each time when it is stored into *E-List*. Generally, *LevelNum* must be a positive value because it stands for a node's depth in a tree; however, *LevelNum* of *e-nodes* could be negative in our proposal. We use Example 1 below to illustrate the generation of *e-nodes*.

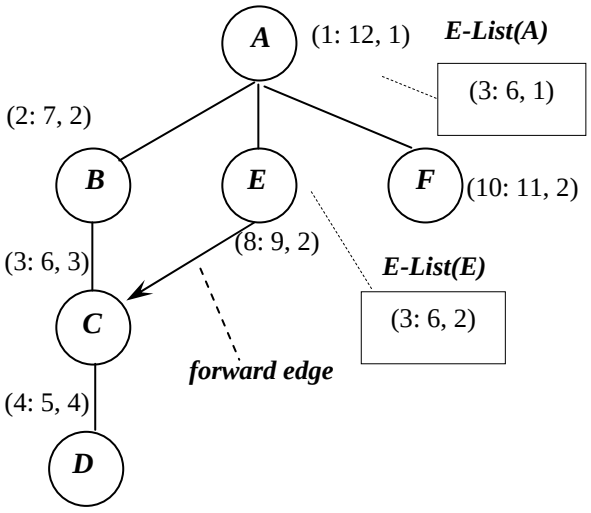

Figure 2. E-List, e-node and forward edge.

Example 1. In Figure 2, edge (*E*, *C*) is a *forward edge*, so *e-node(C)* = (3: 6, 2) is saved into *E-List(E)*. Further, an *e-node* (3: 6, 2–1), i.e., (3: 6, 1), is saved into *E-List(A)* to reflect the fact that we can walk from node *A* to node *D* via the *forward edge (E, C)*. The interval of e-*node* (3: 6, 1) contains that of node *D*, resulting in a path *A* ~ *D* including the *forward edge (E, C)*. Also, if

there is an *e-node(a: b, c)* should be placed in *E-List(C)*, then an *e-node (a: b, c–1)* is saved into *E-List(D)* and an *e-node (a: b, c–2)* into *E-List(A)*.

E-Lists and e-nodes are devised to deal with the multiple-inheritance and cyclic path problems. If there exists any *b/f edge*, the *intervals (S: E)* of ancestor and descendant would fail to determine their structural relationships in traditional approaches. For example, in Figure 2, node *E* is an ancestor of node *D*, but *interval* $(S_E: E_E)$ does not contain *interval* $(S_D: E_D)$. In contrast, *E-nodes* are very useful in dealing with this situation. We insert an *e-node(C)* labeled as (3: 6, 2) into *E-List(E)*, and then node *E* uses the *e-node(C)* to contain node *D*. Certainly, if there exists a path from node *C* to node *Z*, then the path length from *E to Z* is equal to (the path length from *C* to *Z*) + 1. This is the reason why we set $L_{e\text{-}node(C)}$ to $L_C - 1$. When *E* is considered as the ancestor, it makes an "expansion" to include all of its *e-nodes*, meaning that *E* connects to its children not only by *tree edges* but also by *b/f edges*.

We have the following Lemmas for our *Labeled DFS Graph*. Note that in Lemma 1, according to [5], if two intervals intersect, it cannot be the case that one interval is entirely contained in the other one.

Lemma 1. *In a Labeled DFS Graph where nodes are labeled by value (S: E, L), intervals (S: E) do not intersect each other.*
Proof: (excerpt from [5]) Note that for all nodes *u*, $S_u < E_u$. Let *u, v* be two nodes in a *Labeled DFS Graph*, and we begin with the case in which $S_u < S_v$. There are two sub-cases:
(i) If $S_v < E_u$, then *v* was visited while the subtree rooted at *u* was being labeled. This implies that *v* is a descendant of *u*, and *interval* $(S_v: E_v)$ is entirely contained within *interval* $(S_u: E_u)$.
(ii) If $S_v > E_u$, then $S_u < E_u < S_v < E_v$, implying that *intervals* $(S_v: E_v)$ and $(S_u: E_u)$ are disjoint.
The case in which $S_v < S_u$ is proved similarly.

Lemma 2. *In a Labeled DFS Graph, if node u is an ancestor of node v, then the three conditions* $S \le S_v$, $E \ge E_v$, *and* $L < L_v$ *must hold, where* $(S: E, L) \in \{(S: E, L) | (S: E, L) = (S_u: E_u, L_u)$ *or* $(S_e: E_e, L_e), \forall e \in$ *e-nodes in E-List(u)}.*
Proof: If edge (*u, v*) is a *tree edge*, *interval* $(S_u: E_u)$ will contain *interval* $(S_v: E_v)$. Due to the transitivity rule, *interval* $(S_u: E_u)$ will also contain the *interval* $(S_z: E_z)$ for all *v*'s descendants *z*. Hence the three conditions hold.
If edge (*u, v*) is a *b/f edge*, *e-node(v)* will be in *E-List(u)* to assure the three conditions hold, because *v* is an ancestor of *z* and so is *e-node(v).*

Lemma 3. *By restricting that any b/f edge does not appear twice in one specific path, there exists no cyclic path that loops endlessly in the Labeled DFS Graph.*
Proof: *Tree edges* will lead to leaves eventually; and *forward edges* only point to the descendant nodes. Any path comprising *tree edges* and *forward edges* will reach its end for sure. In contrast, only *back edges* cause loops. If they are restricted to traverse only once, in the worst case there exists a very long cyclic path that consumes all *back edges*, and then there will be no loop problem at all.

Lemma 4. *The number of b/f edges in the Labeled DFS Graph equals the summation of [(number of incoming edges of node v) –1] for all nodes v having more than one incoming edge.*
Proof: According to Definition 1, *tree edges* are those edges *(u, v)* if node *v* is first visited by exploring edge *(u, v)*. Thus, if node *v* has more than one incoming edge, then only one of these edges is a *tree edge* (the edge explored while node *v* is first visited) and others are all *b/f edges*.

The combination of multiple *b/f edges*, meaning a path consists of more than one *b/f edge*, could not be managed completely by DFS. Suppose *(v, u)* is a *back edge* during DFS search. It implies that *u* is an ancestor of *v* and *u* is not yet explored completely at that moment, because not all of the children nodes (including node *v*) of node *u* have been explored. According to the transitivity rule, *e-node(u)* is saved into *E-List(v)* and so are *e-nodes* in *E-List(u)*. However since node *u* is not completely explored and thus *E-List(u)* is not constructed entirely, we may lose *e-nodes* that should be saved into *E-List(u)* and thus lose paths, for example, a cyclic path which contains $u \sim v \sim u$. Hence we have to scan the whole *E-Lists* repeatedly until no more *e-nodes* are left. The basic idea of tackling this problem is to insert those *e-nodes* into *E-List(v)* from *E-List(u)* that are not inserted in Procedure DFS. However, this may lead to the cascading insertion of *e-nodes*.

2.2: Node Labeling Scheme over XML Graph

The proposed interval-based node-labeling algorithm, namely G_encoding in Figure 3, works as follows. First a modified DFS procedure is adopted to label each node and compute *E-Lists*. In the DFS, Procedure DFS_Visit explores all edges recursively by marking *b/f edges* and generating *e-nodes*. Function EList_Growing generates more *e-nodes* to handle paths consisting of multiple *b/f edges* and returns a value to indicate if there are *e-nodes* generated. *E-List* uses a flag "Change" (initialized to the *False* value) to indicate if *E-List* is growing or not. EList_Growing follows from Lemma 3 to ensure that no cyclic path loops endlessly, and it scans *E-Lists* several times to ensure no *e-node* is missed.

Consider the example in Figure 2. DFS explores nodes *A, B, C, D, E* and *F* in turn. When exploring node *E*, edge (*E, C*) is classified as *forward edge* (because node *C* was visited), and we put an *e-node(C)* labeled as $(S_C: E_C, L_C - 1)$ = (3: 6, 2) into *E-List(E)* in Step 3.4 of DFS_Visit. Later when DFS finishes examining node *E's* adjacency list, we put all *e-nodes* of *E-List(E)*, say (3: 6, 2–1) = (3: 6, 1), into *E-List(A)* in Step 3.4 of

DFS_Visit. Suppose there is an *e-node (S: E, L)* in *E-List(C)* that does not appear while edge (*E, C*) is explored. Then e-nodes *(S: E, L–1)* and *(S: E, L–2)* will be put into *E-List(E)* and *E-List(A)* respectively in Step 1.2 of EList_Growing. Note that in Steps 3.3 and 3.4 of DFS_Visit and Step 1.2 of EList_Growing, the e-*node*'s *LevelNum* decreases 1 each time when it is stored into the *E-List*.

Algorithm G_encoding(*G*) {
Input: *G* = (*V*, *E*), a rooted directed cyclic graph
Output: *G* (with labeled nodes and *G*'s *E-Lists* containing the complete set of *e-nodes*)
Method:
Step 1: label nodes in G in DFS order by calling DFS(*G*);
Step 2: generate all the *E-Lists* of *G* by calling EList_Growing(*G*) repeatedly till it returns *False*;
}
Procedure DFS(*G*) { // Labeling nodes in G
Input: *G*= (*V*, *E*), a rooted directed cyclic graph
Output: *G* (with labeled nodes and *G*'s *E-Lists*)
Method:
Step 1: mark each node in *G* "not visited" and initialize time-stamp *times* to 0;
Step 2: for each node *u* in *G*, label *u* by calling DFS_Visit(*u*, 1, *times*, *E-List*(*u*)) if *u* was not visited before;
Step 3: return G;
}
Procedure DFS_Visit(*u*, *levels*, *times*, *E-List(u)*) {
Input: node *u*, *u*'s *levels*, time-stamp *times*, *E-List(u)*
Output: *u*, *E-List*(*u*)
Method:
Step 1: mark node *u* "visited";
Step 2: increase *times* by 1, set *u*'s *LevelNum* to *levels* and *u*'s *StartPos* to *times*;
Step 3: for each node *v* adjacent to *u*, if *v* was not visited, goto Step 3.1; otherwise goto Step 3.4;
Step 3.1: mark *u* as *v*'s parent;
Step 3.2: label nodes under *v* by calling DFS_Visit(*v*, *levels*+1, *times*, *E-List*(*u*)) recursively;
Step 3.3: insert the set $\{x \mid x \in E\text{-}List(v)\}$ into *E-List(u)*, goto Step 4;
Step 3.4: insert the set $\{\text{e-node}(v)\} \cup \{x \mid x \in E\text{-}List(v)\}$ into *E-List*(*u*);
Step 4: mark node *u* "done";
Step 5: increase *times* by 1 and set *u*'s *EndPos* to *times*;
}
Function EList_Growing(*G*) {
Input: *G* (with labeled nodes and *G*'s *E-Lists*)
Output: Boolean // return False if none of *E-Lists* grow
Method:
Step 1: if *E-List*(*u*) does not exist, return *False* and goto Step 2; otherwise goto Step 1.1;
Step 1.1: for each *e-node*(*v*) in *E-List*(*u*), if *e-node*(*v*)'s *LevelNum* = *v*'s *LevelNum*–1 and *E-List*(*v*) is changed, then goto Step 1.2; otherwise goto Step 2;
Step 1.2: insert the set $\{x \mid x \in E\text{-}List(v)\}$ into *E-List*(*u*) as well as into *E-List*(*t*) for each *u*'s ancestor node *t*;
Step 1.3: mark *E-List*(*v*) "not changed" and *E-List*(*u*) "changed";
Step 2: remove duplicates in *E-List*;
Step 3: if none of *E-List* are changed, return *False*; otherwise, return *True*;
}

Figure 3. Algorithm G_encoding for node labeling.

2.3: Analysis of G_encoding Algorithm

The first step in Algorithm G_encoding is DFS essentially. Both the time and space complexities are O(|V| + |E|) if graph G = (V, E) is represented by an adjacency list, or $O(|V|^2)$ if graph G is represented by an adjacency matrix. Computing *E-Lists* in procedure DFS_Visit costs $O(N_L * |V| * (N_e + N_e * N_L) / 2)$, where N_e = average number of *e-nodes* in each *E-List* and N_L = average number of nodes associated with an *E-List*. N_L equals to |V| in the worst case, the time and space complexities are $O(|V|^3 * N_e)$. However since *E-Lists* grow during the DFS labeling, it may require more than one pass to scan *E-Lists* until no more *E-List* grows. Although the time and space complexities may grow tremendously in the worst case, it is a one-time cost and, once all nodes have been labeled, the structural join operations (as described in the next Section) can be performed very efficiently hereafter.

3: Structural Join

Our structural join algorithm "SJG", standing for "Structural Join on Graph", is derived from existing structural join algorithms like Tree-Merge-Desc [2] but under the new labeling scheme described in Section 2. Consider an ancestor-descendant relationship *u//v*, and let *A-List* and *D-List* be the two lists of nodes that match the predicates on *u* and *v*, respectively. Taking the two lists as input, Algorithm SJG first expands *A-List*, then outputs all pairs of nodes that match structural relationships in order of descendant's (or ancestor's) *StartPos* by Function Tree-Merge-Desc, and finally removes duplicate results. There are three major

differences between our structural join algorithm and that in [2].

(1) Before Tree-Merge-Desc proceeds, SJG performs an "*expansion*" process which makes all *e-nodes* in *E-List(u),* $\forall u \in$nodes in *A-List,* be included in *A'-List*. This process is necessary since nodes in *A-List* can use their *e-nodes* to join nodes in *D-List.*

(2) With the help of *e-nodes*, SJG uses the conditions in Lemma 2 to determine structural relationships.

(3) Duplicates need to be removed from the result set of node pairs in SJG.

Algorithm SJG(*A-List*, *D-List*) {
Input: *A-List*: the list of potential ancestors sorted in order of StartPos; *D-List*: the list of potential descendants sorted in order of StartPos
Output: *OutputList* // result of Structural Join (of *A-List* and *D-List*) on Graph
Method:
Step 1: expand *A-List* into *A'-List* by calling Expansion(*A-List*);
Step 2: perform structural join by calling Tree-Merge-Desc(*A'-List*, *D-List*) and store the result in *OutputList*;
Step 3: eliminate duplicates in *OutputList*;
}
Function Expansion(*A-List*) {
Input: *A-List*
Output: *A'-List* //a list of node pairs <*u*, *eu*>, where *eu* is an ancestor for join, *u* is for output
Method:
Step 1: generate *A'-List* by collecting *u*, $\forall u \in$nodes in *A-List*, and *e-nodes* in *E-List(u)*;
Step 2: sort node pairs in *A'-List* by the second tuple's *StartPos* and *LevelNum*;
Step 3: return *A'-List*;
}
Function Tree-Merge-Desc(*A'-List*, *D-List*) {
Input: *A'-List*={<*u*, *eu*>}, *D-List*={<*u*>}
Output: *OutputList*
Method:
Step 1: for each node *d* in *D-List*
Step 2: for each node pair <*u*, *eu*> in *A'-List*,
Step 3: if *eu* is an ancestor of *d*, then add <*u*, *d*> into *OutputList*;
Step 4: return *OutputList*;
}

Figure 4. Algorithm SJG for structural join.

3.1: Algorithm SJG

Figure 4 depicts our Algorithm SJG that performs structural joins over the *Labeled DFS Graph*, together with its two major functions Expansion and Tree-Merge-Desc.

In Function Expansion, *A-List* is expanded into *A'-List* whose elements are in the form of node pairs <*output_node*, *join_node*>. *join_node* is used as ancestor for joining with descendants, and *output_node* is for output. For instance, in Figure 5, if *A-List* contains node *G* (12: 13, 2) which has an *e-node* (3: 6, 2), then *A'-List* will include the node pair <(12: 13, 2), (12: 13, 2)> and <(12: 13, 2), (3: 6, 2)> due to Step 1 of Expansion. The node pair <(12: 13, 2), **(12: 13, 2)**> means node *G* **(12: 13, 2)** is used as ancestor for join, while <(12: 13, 2), **(3: 6, 2)**> means the e-node **(3: 6, 2)** is used as ancestor for join. It is necessary to sort *A'-List* in order of *join_node's StartPos* and *LevelNum*, because after expansion, *join_nodes* in *A'-List* may have the same *(S: E)* but different *LevelNum*. In Steps 1 to 3 of Tree-Merge-Desc, *eu* of node pair <*u*, *eu*> in *A'-List* is used to join with node, say *d*, in *D-List*, but the output pair is still <*u*, *d*>, rather than <*eu*, *d*>. In the final step of SJG, duplicates are removed from the result set to keep the final tuple (output pair, path length) unique. In a tree the path exists between two nodes must be unique, but this may not be true in a rooted directed cyclic graph.

Example 2. Suppose there is a regular path query *m//n* as indicated in Figure 5. Let *A-List* = {(2: 11, 2), (12: 13, 2)} and *D-List* = {(4: 5, 4), (8: 9, 4)} corresponding to the two nodes tagged by "*m*" and the other two nodes tagged by "*n*" respectively. *A-List* is first expanded into *A'-List* = {<(2: 11, 2), (2: 11, 2)>, <(12: 13, 2), (3: 6, 2)>, <(2: 11, 2), (8: 9, 3)>, <(12: 13, 2), (12: 13, 2)>}. In Tree-Merge-Desc, the node pair <(2: 11, 2), (8: 9, 3)> uses (8: 9, 3) to join with (8: 9, 4), and generates the output pair <(2: 11, 2), (8: 9, 4)>, rather than <(8: 9, 3), (8: 9, 4)>; the path length is 1, calculated by subtracting the *LevelNum* of (8: 9, 3) from that of (8: 9, 4). Note that node (8: 9, 3) is an *e-node* that belongs to (2: 11, 2) but does not actually exist in the document. All output pairs with path length for this query are shown in Table 1, where there are four paths from *m* to *n* and two of them contain *forward edges*.

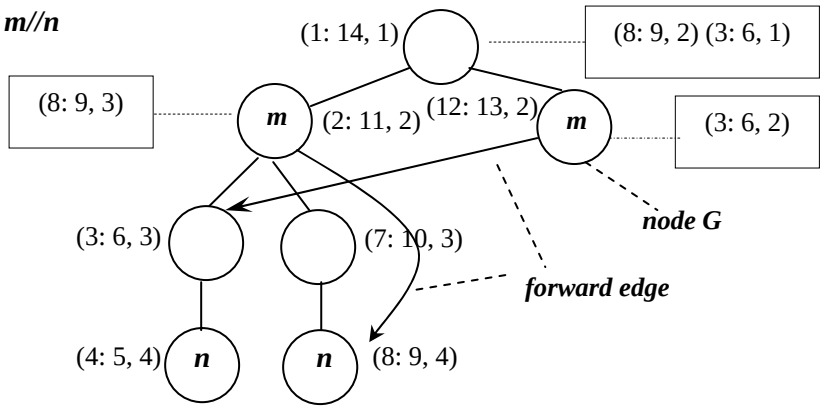

Figure 5. An example of matching ancestor-descendant relationships.

Output Pairs	Path Length
<(2: 11, 2), (4: 5, 4)>	2
<(2: 11, 2), (8: 9, 4)>	2
<(12: 13, 2), (4: 5, 4)>	2
<(2: 11, 2), (8: 9, 4)>	1

Table 1: Query results for Figure 5.

3.2: Analysis of SJG Algorithm

Algorithm SJG's core function is Tree-Merge-Desc. According to [2], its time complexity is O(|*A-List*| + |*D-List*| + |*OutputList*|), and O(|*A-List*| + |*D-List*| + |*OutputList*|^2) in the worst case. If Stack-Tree-Desc is used in SJG instead of Tree-Merge-Desc, its time complexity will be O(|*A-List*| + |*D-List*| + |*OutputList*|). Function Expansion takes O(|*A-List*| * number of *e-nodes* in each *E-List*). The procedure to eliminate duplicates takes O(|*OutputList*| * log(|*OutputList*|)).

4: Conclusions and Future Work

The main contribution of this paper is the solution of matching structural relationships in graph-structured XML data, under a new interval-based labeling scheme. Based on the idea of *e-nodes*, each XML node can be labeled (with an optional *E-List*) appropriately by the G_encoding algorithm. *E-Lists* are devised to solve the multiple-inheritance and cyclic paths problems. The properties of our labeling scheme are formally analyzed. We also develop an efficient structural join algorithm SJG using our labeling scheme. The core function Tree-Merge-Desc in Algorithm SJG can be replaced, if needed, by other existing structural join algorithms easily.

As to the future work of this paper, we plan to reduce the cost of computing *E-Lists* and to shrink the size of intermediate results for each basic binary structural join.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal on Digital Libraries*, Volume 1, Number 1, 1997, pages 68-88.

[2] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y.Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002, pages 141-152.

[3] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002, pages 310-321.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents," *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002, pages 263-274.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, published by MIT Press and McGraw-Hill, 2001.

[6] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "XML-QL: A query language for XML," Available at http://www.w3.org/TR/NOTE-xml-ql/, 1998.

[7] H. V. Jagadish *et al.*, "TIMBER: A Native XML Database," *VLDB Journal: Very Large Data Bases*, Volume 11, Issue 4, 2002, pages 274-291.

[8] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Joins," *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, 2003, pages 253-263.

[9] J. Kim, S. H. Lee, and H.-J. Kim, "Efficient Structural Joins with Clustered Extents," *Information Processing Letters*, Volume 91, Number 2, 2004, pages 69-75.

[10] Q. Li, and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001, pages 361-370.

[11] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001, pages 425-436.