A New Algorithm for Transitive Closures and Computation of Recursion in relational Databases

Yangjun Chen*
Dept. Business Computing, Winnipeg University,
515 Portage Ave. Winnipeg, Manitoba, Canada R3B 2E9
ychen2@uwinnipeg.ca

Abstract *In this paper, we propose a new algorithm for computing recursive closures. The main idea behind this algorithm is tree labeling and graph decomposition, based on which the transitive closure of a directed graph can be computed in $O(e \cdot d_{max} \cdot d_{out})$ time and in $O(n \cdot d_{max} \cdot d_{out})$ space, where n is the number of the nodes of the graph, e is the numbers of the edges, d_{max} is the maximal indegree of the nodes, and d_{out} is the average outdegree of the nodes. Especially, this method can be used to efficiently compute recursive relationships of a directed graph in a relational environment.*

1. Introduction

Let $G = (V, E)$ be a directed graph (*digraph* for short). Digraph $G^* = (V, E^*)$ is the reflexive, transitive closure of G if $(v, w) \in E^*$ iff there is a path from v to w in G. In this paper, we present a new algorithm for computing the transitive closure of a digraph efficiently.

There are several well-known algorithms for computing the transitive closure of a directed graph [Wa62, Eb81, Sc83, Wa75, IRW93]. All of them have, however, the worst-case time complexity $O(n \cdot e)$, where n is the number of the nodes of the digraph, and e is the number of the edges. These methods were further improved by some other researchers such as the algorithms proposed by Italiano [It86], and La Poutre and Leeuwen [LL88]. Their algorithms maintain the transitive closure of a directed graph in $O(n \cdot e)$ time and $O(n^2)$ space. In 1997, Abdeddaim proposed an algorithm to compute transitive closures incrementally, which is especially suited for *alignment of DNA sequences* [Ab97]. This algorithm, however, needs knowing a *spanning set* of disjoined paths (each node is in one and only one path), which cover all the nodes of a digraph. Based on the knowledge of the spanning set, this algorithm can compute the transitive closure of a digraph in $O(k^2 \cdot e + n \cdot \min\{n, e\})$ time and $O(k \cdot n)$ space, where k is the number of disjoined paths and is equal to the breadth of the digraph.

Recently, the implementation of transitive closures in a disk-based environment has been received an extensive attention. A lot of researches have been directed to the efficient implementation and performance evaluation using or modifying the classical algorithms and different data structures aiming at the reduction of I/O traffic [ADJ90, AJ89, AJ90, DR94, IRW93, Ji90]. However, from an algorithmic point of view, the performance of transitive closures has not been improved *per se* since it is limited by the algorithms used. In 1996, Teuhola proposed a method to encode ancestor-descendant relationship and to speed-up recursion in relational databases [Te96]. However, this encoding method cannot be used to develop efficient algorithm for computing recursive closures. Moreover, in the case of acyclic digraphs (DAGs), a graph needs to be decomposed into a series of trees; but no formal decomposition was proposed in [Te96].

In this paper, we introduce a new way to compute transitive closures of acyclic digraphs based on *tree labeling* and *graph decomposition*. Instead of storing all the descendants for a node in the result, we store a set of nodes S in the decomposed trees; and each node v in S represents a subtree rooted at v in one of the decomposed trees. In this way, the transitive closure of a DAG G can be stored using $O(n \cdot d_{max} \cdot d_{out})$ space, where d_{max} is the maximal indegree of the nodes, and d_{out} is the average outdegree of the nodes. Accordingly, the time complexity can be reduced to $O(e \cdot d_{max} \cdot d_{out})$. This computational complexity is superior to any existing ones. In addition, this method can be applied to a digraph containing cycles by using Tarjan's algorithm for identifying *strongly connected components* (SCCs) [Ta72].

The rest of the paper is organized as follows. In Section 2, we discuss tree labeling and graph decomposition, which are needed to develop our algorithm. In Section 3, we show a new strategy for computing recursive closures in great detail. In Section 4, we discuss briefly how to use graph labeling to speed-up recursion in relational databases. Section 5 reports test results. Finally, a short conclusion is set forth in Section 6.

2. Tree labeling and graph decomposition

In this section, we mainly discuss the concepts of tree labeling and graph decomposition, based on which our algorithm is designed.

For any directed tree T, we can label it as follows.

By traversing T in *preorder*, each node v will obtain a number $pre(v)$ to record the order in which the nodes of the tree are visited. In a similar way, by traversing T in *postorder*, each node v will get another number $post(v)$. These two numbers can be used to characterize the ancestor-descendant relationship as below.

Proposition 1. Let v and v' be two nodes of a tree T. Then, v' is a descendant of v iff $pre(v') > pre(v)$ and $post(v') < post(v)$.

Proof. See [Kn73]. □

The following example helps for illustration.

Example 1. See the pairs associated with the nodes of the directed tree shown in Fig. 1. The first element of each pair is the preorder number of the corresponding node and the second is its postorder number. Using such labels, the ancestor-

* The author is supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada).

descendant relationship can be easily checked.

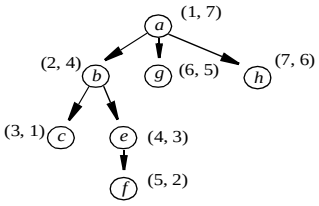

Fig. 1. Labeling a tree

For instance, by checking the label associated with *b* against the label for *f*, we know that *b* is an ancestor of *f* in terms of Proposition 1. We can also see that since the pairs associated with *g* and *c* do not satisfy the condition given in Proposition 1, *g* must not be an ancestor of *c* and *vice versa*.

For two pairs (p, q) and (p', q') associated with *u* and *v*, respectively, we say that (p, q) is subsumed by (p', q'), denoted $(p, q) \prec (p', q')$, if $p > p'$ and $q < q'$. Then, *u* is a descendant of *v* if (p, q) is subsumed by (p', q').

For a given acyclic digraph (DAG) *G*, we want to decompose it into a series of directed trees so that the above property can be used to speed up the computation of transitive closures.

First, we choose arbitrarily a root r_1 (a node with *indegree* = 0) in *G* and traverse *G* from r_1 in such a way that each encountered node is accessed only once. This can be done by searching *G* in a depth-first manner and marking the nodes when they are first encountered. We denote such a tree $T_{max}(G)$. Then, we remove $T_{max}(G)$ and subsequently all isolated nodes from *G*, getting another digraph $G_1 = (V_1, E_1)$. Next, we construct a directed tree w.r.t. G_1: $T_{max}(G_1)$. We repeat this process until the remaining graph becomes empty. It is therefore easy to see that all $T_{max}(G_i)$'s can be obtained in $O(m(n + e))$ time by repeating graph search procedure *m* times, where *m* is the maximal indegree of the nodes. However, this time complexity can be reduced to $O(n + e)$ by implementing an algorithm which computes such a sequence in a single-scan.

For a DAG $G_0 = (V_0, E_0)$, we represent the sequence of the directed trees $T_{max}(G_i)$ $(i = 0, 1, ..., m - 1)$ as follows:

$$T_{max}(G_0) = (V_1, E_1),\ G_1 = G_0 - T_{max}(G_0),$$
$$T_{max}(G_1) = (V_2, E_2),\ G_2 = G_1 - T_{max}(G_1),$$
$$\text{... ...}$$
$$T_{max}(G_{m-2}) = (V_{m-1}, E_{m-1}),\ G_{m-1} = G_{m-2} - T_{max}(G_{m-2}),$$
$$T_{max}(G_{m-1}) = (V_m, E_m),\ G_m = G_{m-1} - T_{max}(G_{m-1}) = \varnothing;$$

where $T_{max}(G_i)$ $(i = 1, ..., m - 1)$ represents a directed tree obtained by traversing G_i from a root r_i in G_i, and *k* is the largest indegree of the nodes of G_0.

In the following, we give a linear time algorithm to compute all $T_{max}(G_i)$'s.

The main idea is to construct all $E_1, E_2, ... E_m$ in a single scan. During the graph search we compute, for each edge *e* being scanned, the *i* satisfying $e \in E_i$. Such *i* can be defined to be the smallest such that if *e* is put in E_i, the condition: each node in any E_j $(j = 1, ..., i)$ is visited only once, is not violated, where E_i denotes the edge sets constructed so far. In the algorithm, we always choose an unvisited edge *e* that is adjacent to edge $e' \in E_i$ with the largest *i*. To do that, we associate each node *v* with a label $l(v)$: $l(v) = i$ indicates that *v* has been reached by an edge of the tree $T_{max}(G_{i-1}) = (V_i, E_i)$. In the following algorithm, we assume that the nodes are numbered in terms of the depth-first search.

Algorithm *find-forest*
input: $G = (V, E)$
output: $E_1, E_2, ..., E_m$
begin
  $E_1 := E_2 := ... := E_m := \varnothing$;
  Mark all nodes $v \in V$ and all edges $e \in E$ "unvisited";
  $l(v) := 0$ for all $v \in V$;
  while there exist "unvisited" nodes **do**
    begin
      choose an "unvisited" node $v \in V$ with the largest l and the smallest "depth-first" number;
      for each "unvisited" edge *e* incident to *v* **do**
        begin
          Let *u* be the other end node of *e* $(\neq v)$;
* $E_{l(u)+1} := E_{l(u)+1} \cup \{e\}$;
** $l(u) := l(u) + 1$;
***   **if** $l(v) < l(u)$ **then** $l(v) := l(u) - 1$;
          Mark *e* "visited";
        end
      Mark *x* "visited";
    end
end

For example, by applying the above algorithm to the graph shown in Fig. 2(a), we will obtain three directed trees shown in Fig. 2(b). In Appendix A, we will trace the execution of the algorithm against Fig. 2(a) for a better understanding.

In the above algorithm, each edge is visited exactly once. Therefore, the time complexity of the algorithm is bounded by $O(n + e)$. In the following, we prove a proposition to establish the correctness of the algorithm.

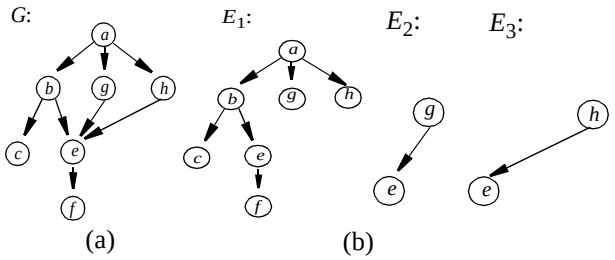

Fig. 2. DAG and its node-disjunct maxiaml trees

Proposition 2. Applying Algorithm *"find-forest"* to a DAG *G*, a sequence of directed trees w.r.t. *G* will be found, which covers all of its edges.

Proof. First, we note that by the algorithm each edge will be visited exactly once and put in some E_i. Therefore, the union of all E_i's will contains all edges of *G*. To prove the proposition, we need now to specify that in every E_i, except the root of E_i, each node can be reached along only one path, or say, visited exactly one time w.r.t. E_i. Pay attention to the lines marked with * and **. If a node *u* is visited several times along different edges, such edges will be put in different E_i's. Therefore, in each E_i, *u* can be visited only once. By the line marked with ***, if an edge (v, u) is put in some E_i, then an unvisited edge reaching *v* afterwards will be put in E_i or E_{i+1}. If in E_i there is no edge reach *v* up to now (in this case, $l(v) < l(u)$ holds), the label of *v* will be changed to $i - 1$. Then, if afterwards an unvisited edge reaches *v*, it will be put in E_i. Otherwise, $l(v) = l(u)$ and there must already be an edge in E_i reaching *v*. Thus, if afterwards an unvisited edge reaches *v*, it will be put in E_{i+1}. In this way, in E_i, *v* can be visited only

once, which completes the proof. □

Now we can label each E_i in the same way as discussed in the previous section. In addition, we notice that a node may appear in several E_i's. For example, in Fig. 2(b) node g appears in E_1 and E_2 while node e occurs in all the three directed trees. Then, after labeling each E_i, each node v will get a pair sequence of the form: $(pre_{i_1}, post_{i_1})$. $(pre_{i_2}, post_{i_2})$. ... $(pre_{i_j}, post_{i_j})$, where for each $i_k \in \{1, ..., m\}$ (m is the largest indegree of the nodes) $(pre_{i_k}, post_{i_k})$ stands for the preorder number and postorder number of v w.r.t E_{i_k}.

Example 2. The directed trees shown in Fig. 2(b) can be labeled as shown in Fig. 3(a). Then, each node in the digraph shown in Fig. 2(a) is associated with a sequence of pairs as shown in Fig. 3(b).

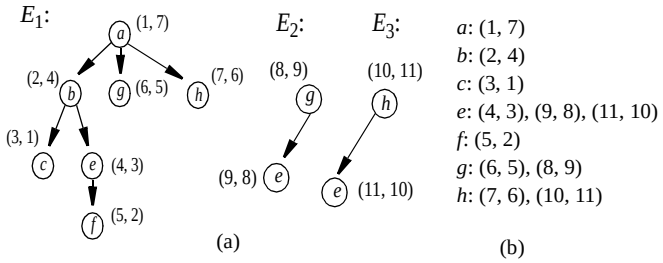

Fig. 3. DAG labeling

Note that the integers for numbering E_i start from $|V_{i-1}| + ... + |V_1| + 1$. For instance, in Fig. 3(a), E_2 is numbered using integers from 8 and E_3 is from 10.

3. Computation of transitive closure

Now we begin to discuss how to compute transitive closures using the pair sequences associated with the nodes. First, we consider DAGs in 3.1. Then, the case of digraphs containing cycles is discussed in 3.2.

3.1 Transitive closure of DAGs

As with labels for a tree, what we want is to use the sequences of pairs to identify ancestor-descendant relationships in a DAG. Assume that (p_1, q_1), ..., (p_g, q_g) and (p_1', q_1'), ..., (p_h', q_h') are two sequences of pairs associated with v and v', respectively. If there exist i and j ($1 \le i \le g$, $1 \le j \le h$) such that $(p_i, q_i) \prec (p_j', q_j')$, then v is a descendant of v'. To find correct pair sequences for the nodes in a DAG $G = (V, E)$, we sort the nodes topologically, i.e., $(v_i, v_j) \in E$ implies that v_j appears before v_i in the sequence of the nodes. We scan the topological sequence of the nodes from the beginning to the end and merge, at each step, the pair sequence of each of a node's children into its pair sequence. To speed up the merging operation, we stored the pairs for a node v in a link list A_v.

Let v be the node being considered. Let v_1, ..., v_k be the children of v. Merge A_v with each A_{v_i} (i = 1, ..., k) as follows. Assume that $A_v = (p_1, q_1) \rightarrow ... \rightarrow (p_g, q_g)$ and $A_{v_i} = (p_1', q_1') \rightarrow ... \rightarrow (p_h', q_h')$ are two pair sequences stored as link lists as shown in Fig. 4. Assume that both A_v and A_{v_i} are increasingly ordered. (We say a pair p is larger than another pair p', denoted $\alpha > \beta$ if $\alpha.pre > \beta.pre$ and $\alpha.post > \beta.post$.)

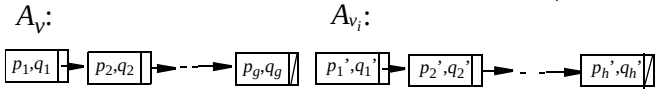

Fig. 4. Link lists associated with nodes in G

We step through both A_v and A_{v_i} from left to right. Let (p_i, q_i) and (p_j', q_j') be the pairs encountered. We'll make the following checkings.

(1) If $p_i > p_j'$ and $q_i > q_j'$, insert (p_j', q_j') into A_v after (p_{i-1}, q_{i-1}) and before (p_i, q_i), and move to (p_{j+1}', q_{j+1}').

(2) If $p_i > p_j'$ and $q_i < q_j'$, remove (p_i, q_i) from A_v and move to (p_{i+1}, q_{i+1}). ((p_i, q_i) is subsumed by (p_j', q_j').)

(3) If $p_i < p_j'$ and $q_i > q_j'$, ignore (p_j', q_j') and move to (p_{j+1}', q_{j+1}'). ((p_j', q_j') is subsumed by (p_i, q_i).)

(4) If $p_i < p_j'$ and $q_i < q_j'$, ignore (p_i, q_i) and move to (p_{i+1}, q_{i+1}).

(5) If $p_i = p_j'$ and $q_i = q_j'$, ignore both (p_i, q_i) and (p_j', q_j'), and move to (p_{i+1}, q_{i+1}) and (p_{j+1}', q_{j+1}'), respectively.

In terms of the above discussion, we have the following algorithm to merge two pair sequences together.

Algorithm *pair-sequence-merge*(A_1, A_2)

Input: A_1 and A_2 - two link lists associated with v_1 and v_2.

Output: A - modified A_1, obtained by merging A_2 into A_1, containing all the pairs in A_1 and A_2 with all the subsumed pairs removed.

```
begin
1   p ← first-pair(A1);
2   q ← first-pair(A1);
3   while p ≠ nil do{
4    while q ≠ nil do{
5      if (p.pre > q.pre ∧ p.post > q.post) then
6          {insert q into A1 before p;
7          q ← q.next;}
8          else if (p.pre > q.pre ∧ p.post < q.post) then
9          {temp ← p; (*p is subsumed by q; remove p from A1.*)
10         remove p from A1;
11         p ← temp.next;}
12     else if (p.pre < q.pre ∧ p.post > q.post) then
13         {q ← q.next;}
               (*q is subsumed by p; move to the next element of q.*)
14           else if (p.pre < q.pre ∧ p.post < q.post) then
15               {p ← p.next;}
16               else if (p.pre = q.pre ∧ p.post = q.post) then
17                    {p ← p.next; q ← q.next;}
16  if p = nil ∧ q ≠ nil then {attach the rest of A2 to the end of A1;}
end
```

In the following, we establish several propositions to clarify the properties of the above algorithm. We say a pair p is larger another pair p', denoted $p > p'$ if $p.pre > p'.pre$ and $p.post > p'.post$.

Proposition 3. Let A_1 and A_2 be two pair sequences sorted in increasing order. Let A be the result of merging A_2 into A_1 using Algorithm *pair-sequence-merge*. Then, A is also sorted in increasing order.

Proof. During the execution of the algorithm, some pairs may be removed from A_1 and some pair of A_2 may be inserted into A_1. Let q be a pair of A_2 inserted into A_1. It may be inserted into A_1 in line 6 or in line 18. If it is inserted into A_1 in line 6, there must be pair p in A_1 such that $p > q$. Consider the pair p' before p. We have $p' < q$; otherwise, q will be inserted before p' or will not be inserted A_1 into at all. In this case, the proposition holds. If q is inserted into A_1 in line 18, all the pairs in

A_1 must be used up before this line is executed. We notice that at this moment, all the pairs in A_1 are increasingly ordered and smaller than all the remaining pairs in A_2, which are also increasingly ordered. Therefore, in this case, the proposition holds, too. □

Proposition 4. Let A_1 and A_2 be two pair sequences sorted in increasing order. Let A be the result obtained by merging A_2 into A_1 using Algorithm *pair-sequence-merge*( ). If v is a node in a subtree of G_r, which is rooted at some node labeled with a pair in A_2, then there must be a pair in A such that the subtree rooted at it contains v.

Proof. Assume that v is in a subtree rooted at u labeled with (*pre*, *post*) that appears in A_2. If (*pre*, *post*) appears in A, the proposition holds. Suppose that (*pre*, *post*) does not appear in A. In this case, there must be a pair (*pre*', *post*') in A_1, which subsumes (*pre*, *post*). Notice that (*pre*', *post*') cannot be subsumed by any pair in A_2 since it subsumes (*pre*, *post*). Otherwise, we will have a pair (*pre*'', *post*'') in A_2 such that *pre*'' < *pre*' and *post*'' > *post*'. But we have (*pre*, *post*) < (*pre*'', *post*'') or (*pre*, *post*) > (*pre*'', *post*''). In the former case, we have *pre* < *pre*'' < *pre*'. It contradicts the fact that (*pre*', *post*') subsumes (*pre*, *post*). In the latter case, we have *post* > *post*'' > *post*'. It also contradicts the fact that (*pre*', *post*') subsumes (*pre*, *post*). Therefore, (*pre*', *post*') will appear in A. Since v is in the subtree rooted at (*pre*, *post*), it must be in the subtree rooted at (*pre*', *post*'). Thus, the proposition holds.

Proposition 5. Let A_1 and A_2 be two pair sequences sorted in increasing order. Let A be the result obtained by merging A_2 into A_1 using Algorithm *pair-sequence-merge*( ). If v is a node in a subtree of G_r, which is rooted at some node labeled with a pair in A_1, then there must be a pair in A such that the subtree rooted at it contains v.

Proof. Similar to Proposition 3. □

Proposition 6. The time complexity of Algorithm *pair-sequence-merge* is bounded by O(max{$|A_1|$, $|A_2|$}).

Proof. During the execution of the algorithm, each pair in A_1 and A_2 is visited at most once. □

Based on the merging operation, the pair sequences for all the nodes of a DAG can be computed using the following algorithm.

Algorithm *transitive-closure*
begin
1 Let v_n, v_{n-1}, ..., v_1 be the topological sequence of the nodes of G;
2 **for** i **from** n downto 1 **do**
3 {let v_{i_1}, ..., v_{i_k} be the child nodes of v_i;
4 **for** j **from** 1 to k **do**
5 call *pair-sequence-merge*(A_i, A_j);
6 }
end

Example 3. A possible topological sequence of the exemplary digraph is shown in Fig. 5(a). Each of the nodes in the sequence is associated with a sequence of pairs that are obtained by labeling the trees shown in Fig. 2(b). Applying Algorithm *transitive-closure* to the topological sequence, each node will be associated with a pair sequence as shown in Fig. 5(b).

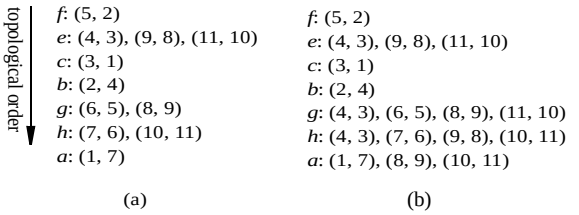

Fig. 5. Pair sequence generation along a topological order

The above algorithm can be considered as a new way to compute transitive closures with a different representation of results. In a traditional method, a result is represented as the set of all the nodes of a digraph with each associated with a set containing all its descendants. In our method, the result is also the set of all nodes of a digraph, but with each associated with a set of pointers to some subtrees rooted at some nodes in decomposed directed trese.

Now we consider the computational complexity of the proposed algorithm. First, we have the following Proposition.

Proposition 7. For a DAG, the length of a pair sequence associated with a node v of DAG is bounded by $d \cdot m$, where d is the outdegree of v and m is the largest indegree of the nodes of the DAG, respectively.

Proof. First, we note that a DAG can be decomposed into m directed trees, where m represents the largest indegree of the nodes of the DAG. Assume that during the merging process, a node v gets a pair sequence of length $d \cdot m$, in which a pair is not subsumed by anyone else Then, we have exactly d pairs w.r.t. each decomposed tree in the pair sequence. Assume that a new pair is being inserted into the pair sequence. This pair must be subsumed by one of the pairs in the sequence or it subsumes one of them. Therefore, the numbers of pairs associated with v cannot be larger than $d \cdot m$.

Based on the above observation, we have the following two propositions.

Proposition 8. The time complexity of Algorithm *transitive-closure* is bounded by $O(e \cdot d_{max} \cdot d_{out})$, where e is the number of the edges of the DAG, d_{max} is the maximal indegree of the nodes, and d_{out} are the average outdegree of the nodes.

Proof. During the execution of the algorithm, each node in the topological sequence is visited exactly once. To generate the pair sequence for a node v_i, d_i merging operations will be performed, where d_i represents the outdegree of v_i. In terms of Proposition 3, the length of a pair sequence is bounded by k. Therefore, the time complexity of Algorithm *transitive-closure* is bounded by

$$O(\sum d_i \cdot d_{max} \cdot d_i) = O(e \cdot d_{max} \cdot d_{out}) \quad \square$$

Proposition 9.[i] The space complexity of Algorithm *transitive-closure* is bounded by $O(n \cdot d_{max} \cdot d_{out})$, where e is the number of the edges of the DAG, d_{max} is the maximal indegree of the nodes, and d_{out} are the average outdegree of the nodes.

Proof. It is obvious. □

Since the decomposition and the labeling of a DAG need only $O(e)$ time and $O(e)$ space, the whole time complexity of our algorithm is bounded by $O(e \cdot d_{max} \cdot d_{out})$ according to Proposition 8, and the space complexity is bounded by $O(n \cdot d_{max} \cdot d_{out})$ according to Proposition 9. This computation-

al complexity is superior to any existing ones.

Proposition 10. Let v be a node G. Any descendant u of v must be in a subtree of G_r rooted at a node labelled with a pair in A_v constructed by Algorithm *transitive-closure*.

proof. Assume that $v_n \rightarrow v_{n-1} \rightarrow ... \rightarrow v_1$ is a topological sequence of G. We prove the proposition by induction on the ordinal number m in the topological sequence of the nodes.

Basis. When $m = 1$, v_n is a leaf node in G and its link list contains only one label associated with v_n. The proposition holds.

Hypothesis. Suppose that when $m \leq k$ the proposition holds. That is, each link list A_i associated with v_i ($i = n, ..., n - k$) contains all the pairs covering all the descendants of v_i.

Consider $m = k + 1$. According to the property of the topological sequence, all the child nodes of v_{n-k-1} must appear in $\{v_n, v_{n-1}, ..., v_{n-k}\}$. Then, from lines 3 - 6 of Algorithm *transitive-closure*, as well as Proposition 2, 3 and 4, we can see that the link list A_{n-k-1} associated with v_{n-k-1} must contain all the pairs covering all the descendants of v_{n-k-1}. It completes the proof. □

3.2 Transitive closure of cyclic graphs

Based on the method discussed in the previous subsection, we can easily develop an algorithm to compute transitive closure for digraphs containing cycles. First, we use Tarjan's algorithm for identifying strongly connected components (*SCCs*) to find cycles of a cyclic graph [Ta72] (which needs only $O(n + e)$ time). Then, we take each SCC as a single node (i.e., condense each SCC to a node) and transform a cyclic graph into a DAG. Obviously, applying the algorithm *find_forest*( ) to this DAG, we will get a set of directed trees. For each of these trees, we can associate each of its nodes with a pair as discussed above. In this way, all nodes in an SCC will be assigned the same pair (or the same pair sequence).

Finally, we notice that the time complexity of the algorithm for handling cycles remains $O(e \cdot d_{max} \cdot d_{out})$ since Tarjan's algorithm runs in $O(n + e)$ time [Ta72].

4. Computing recursion in relational databases

The transitive closure computed using the above algorithm is stored as a set of separated trees with a set of pointers associated with each node in a digraph. Then, the union of all the trees pointed by the pointers associated with a node contains all the descendants of the node. In addition, this algorithm hints a new way to speed-up recursion in a relational database.

We define the following relational schema:

Node(Node_id, label_sequence1, label_sequence2, Node_rest),

where Node_id is used for node identifiers, label_sequence1 for the label pair sequences obtained by labeling the decomposed trees, label_sequence2 for the label pair sequences constructed by running Algorithm *transitive-closure*, and Node_rest for the other information related to the nodes of a digraph. Then, to retrieve the descendants of node x, we issue two queries. The first query is very simple as shown below:

```
Q1:   SELECT   label_seqnence2
      FROM     Node
      WHERE    Node_id = x
```

Let the label sequence obtained by evaluating the above query be y. Then, the second query will be of the following form:

```
Q2:   SELECT   *
      FROM     Node
      WHERE    F(label_sequence1, y),
```

where $F(s_1, s_2)$ is a boolean function. If there exists a pair p in s_1 and a pair q in s_2 such that $p \prec q$, then $f(s_1, s_2)$ returns *true*; otherwise *false*. The function F can be implemented as follows.

Let $s_1 = (p_1, q_1), ..., (p_k, q_k)$ and $s_2 = (p_1', q_1'), ..., (p_l', q_l')$ be two sequences of pairs associated with v and v', respectively. To see whether v is subsumed by v' or *vice versa*, we do the following checkings when we step through both the sequences from left to right. Let (p_i , q_i) and (p_j', q_j') be the pairs encountered.

(1) If $p_i > p_j'$ and $q_i > q_j'$, move to (p_{j+1}', q_{j+1}').

(2) If $p_i > p_j'$ and $q_i < q_j'$, v is subsumed by v', and return *true*.

(3) If $p_i < p_j'$ and $q_i > q_j'$, v' is subsumed by v, and return *false*.

(4) If $p_i < p_j'$ and $q_i < q_j'$, move to (p_{i+1}, q_{i+1}).

(5) If $p_i = p_j'$ and $q_i = q_j'$, return *true*.

(6) If all the above conditions are not satisfied, return *false*.

From this, we can see that each execution of F function needs only $O(max\{k, l\})$ time.

In comparison with Teuhola's method [Te96], our method has the following advantages.

(1) In Teuhola's method, each node in a tree is associated with an interval $[\alpha, \beta]$ in such a way that the interval $[\alpha', \beta']$ of any descendant of the node belongs to its interval, where α, β, α'or β' is a long bit string, called *signature*, which is produced using a a set of hash functions. Therefore, there exists the so-called signature conflict problem. Such a problem does not exist in our method.

(2) In the case of trees, the length of signatures changes according to the outdegrees of the nodes and the height of the trees. In our method, the label for a node is always a pair of integers.

(3) In the case of DAGs, a digraph needs to be decomposed into a series of 'spanning' trees to use Teuhola's method; but no formal method is proposed to conduct such a decomposition in [Te96]. In our method, the graph decomposition is explicitly defined.

5. Experiment results

We have implemented a test bed in C++, with our own buffer management (with first-in-first-out replacement policy) and B+-tree structure. The computer was Intel Pentium III, running standalone.

We have tested two methods: the method based on signatures (Teuhola's [Te96]), and the method based on graph labeling (the one discussed in this paper).

We used two structure types: Trees and DAGs, and measured the physical I/O quota as well as the cpu time. We did not check cyclic digraphs since in [Te96] no formal method was suggested to handle this case using signatures. In fact, even for DAGs, it was not explicitly discussed in [Te96] how to decompose a digraph into a set of 'spanning' trees (as defined

in the article). We code for the specific data setting described in [Te96]; but it is not a general strategy.

As done in [Te96], we first test the following two cases:

(1) a forest of 18 trees with three children per nonleaf; eight levels, 59040 nodes, and 59022 connections;

(2) a DAG of 640 roots with three children per nonleaf; two parents per nonroot, eight levels, 31525 nodes and 61770 connections.

For the two methods tested, the data files are designed a little bit differently as shown in Fig. 6. In both the data files, a node is represented by a node identifier that is in fact an integer represented as a bit sequence. The file for testing Teuhola's method contains, for each node, 32 bits for its signature plus 4 bits for the level, at which the node appears. In this file, only the low value (a signature) of the interval associated with a node is stored; and the high value of the interval can be calculated using the low value and the corresponding level number. The file for testing the graph-labeling method contains a preorder number and a postorder number for each node. Each of them is 18 bits long.

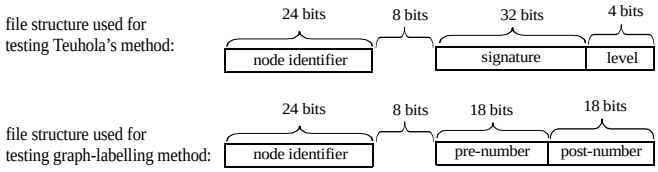

Fig. 6. File structures

The results are gathered in Table 1.

Table 1: Test results

structure	node accessed	Teuhola's page accesses	Teuhola's cpu time (sec.)	graph-labeling page accesses	graph-labeling cpu time (sec.)
Forest	3280	117	3.5	117	2
DAG	3041	1851	39	322	9

From Table 1, we can see that in the case of trees, if the signatures of Teuhola's method have the same length as the labels of the graph-lebeling method, they have almost the same performance. However, in the case of DGAs, Teuhola's method is much worse than ours. It is because one has to decompose a DAG into a series of spanning trees to use Teuhola's method. For each decomposed tree, the signatures associated with the nodes can be used to check descendants; but the child signature of each accessed connection must be checked. If it is outside the [Low, High] range of the signatures of the current tree, then the child belongs to another tree, and a new query is issued against it. This leads to a lot of extra page accesses. For the graph-labeling method, we notice that in the DAG case, although the number of page accesses of this method is not much larger than the case of trees, the time difference of these two cases is relatively big. It is because for a DAG each node is associated with a sequence of label pairs and each check of label pair sequences needs more time.

The following case is tested to see the impact of the outdegrees of the nodes in a tree to the performance.

3) a forest of 18 trees with 4 children per nonleaf; eight levels, 1197648 nodes.

For this case, the file for testing Teuhola's method contains, for each node, 64 bits for its signature plus 4 bits for the level. In the file for testing the graph-labeling method, the preorder number and the postorder number for each node are 22 bits long, respectively.

The results are shown in Table 2.

Table 2: Test results

structure	node accessed	Teuhola's page accesses	Teuhola's cpu time (sec.)	graph-labeling page accesses	graph-labeling cpu time (sec.)
Forest	66536	3746	79	2726	41

From Table 2, we can see that in the case of large outdegrees, signatures become longer and the performance of Teuhola's degrades. It also shows the main drawback of Teuhola's method: sensitivity to the outdegree of nodes. If at some level of a tree there exists a node with a large outdegree, one has to use a long bit string to code the signature pieces for all the nodes at that level. If each level has a node with a large outdegree, the signatures associated with the nodes of the tree must be long enough to differentiate from each other. In contrast, the length of the codes for our labels depends only on the number of the nodes in a graph. Therefore, the code length for integer labels is averagely shorter than signatures. Fig. 7 shows the times elapsed to compute the recursion of a root in a forest containing 18 trees containing 59040. We change the outdegree of the nodes for each run and adjust the signatures so that each time the signatures have different length. In contrast, the label length of our algorithm remains unchanged. This arrangement is reasonable since the length of the nodes' signatures at a level (of a tree) depends on the largest outdegree at this level. If there is a node at a level has a large outdegree, the signatures for all the nodes at that level must be set very long according to the signature construction proposed in [Te96]. However, the length of labels used in ours depends only on the labels' values. The largest label is equal to the number of the nodes in a graph.

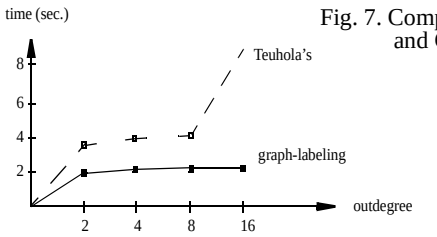

Fig. 7. Comparison of Teuhola's and Graph-labeling

6. Conclusion

In this paper, a new technique for labeling a digraph has been proposed. Using this technique, the recursion w.r.t. a tree hierarchy can be computed very efficiently. In addition, we have devised an algorithm to decompose a DAG into a series of directed trees, which requires only linear time. Together with the labeling technique, this method enables us to develop an efficient algorithm to compute recursive closures for DAGs as well as digraphs containing cycles in $O(e \cdot d_{max} \cdot d_{out})$ time and $O(n \cdot d_{max} \cdot d_{out})$, where e is the number of the edges of a digraph, n is the number of the nodes, d_{max} is the maximal indegree of the nodes, and d_{out} is the average outdegree of the

nodes. Especially, this method hints a new approach to materialize transitive closures in relational databases.

References

Ab97 S. Abdeddaim, On Incremental Computation of Transitive Closure and Greedy Alignment, in: *Proc. 8th Symp. Combinatorial Pattern Matching*, ed. Alberto Apostolico and Jotun Hein, 1997, pp. 167-179.

ACCM97 S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, J. Simon, "Quering documents in object databases," *Int. J. Digital Libraries*, Vol. 1, No. 1, April 1997, pp. 5 - 19.

ADJ90 R. Agrawal, S. Dar, H.V. Jagadish, "Direct transitive closure algorithms: Design and performance evaluation," *ACM Trans. Database Syst.* 15, 3 (Sept. 1990), pp. 427 - 458.

AJ89 R. Agrawal and H.V. Jagadish, "Materialization and Incremental Update of Path Information," in: *Proc. 5th Int. Conf. Data Engineering*, Los Angeles, 1989, pp. 374 - 383.

AJ90 R. Agarawal and H.V. Jagadish, "Hybrid transitive closure algorithms," In *Proc. of the 16th Int. VLDB Conf.*, Brisbane, Australia, Aug. 1990, pp. 326 -334.

BKKG88 J. Banerjee, W. Kim, S. Kim and J.F. Garza, "Clustering a DAG for CAD Databases," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 14, No. 11, Nov. 1988, pp. 1684 - 1699.

BR86 F. bancihon and R. Ramakrishnan, "An Amateurs Introduction to Recursive Query Processing Strategies," in: *Proc. ACM SIGMOD Conf.*, Washington D.C., 1986, pp. 16 - 52.

Ca90 M. Carey et al., "An Incremental Join Attachment for Starburst," in: *Proc. 16th VLDB Conf.*, Brisbane, Australia, 1990, pp. 662 - 673.

CA98 Y. Chen, K. Aberer, "Layered Index Structures in Document Database Systems," *Proc.* 7th *Int. Conference on Information and Knowledge Management (CIKM)*, Bethesda, MD, USA: ACM, 1998, pp. 406 - 413.

CA99 Y. Chen and K. Aberer, "Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases," in: *Proc. of 10th Int. DEXA Conf. on Database and Expert Systems Application*, Florence, Italy: Springer Verlag, Sept. 1999. pp. 473 - 484.

Ch02 Y. Chen, "On the Graph Traversal and Linear Binary-chain Programs," to appear in *IEEE Transactions on Knowledge and Data Engineering*, 2002.

CS92 R.G.G. Cattell and J. Skeen, "Object Operations Benchmark," *ACM Trans. Database Systems*, Vol. 17, no. 1, pp. 1 -31, 1992.

Da86 P. Dadam et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierachies," *Proc. ACM SIGMOD Conf.*, Washington D.C., 1986, pp. 356-367.

DR94 S. Dar and R. Ramarkrishnan, "A Performance Study of Transtive Closure Algorithm," in *Proc. of SIGMOD Int. Conf.*, Minneapolis, Minnesota, USA, 1994, pp. 454 - 465.

Dz75 J. Dzikiewicz, "An Algorithm for Finding the Transitive Closure of a Digraph," *Computing* 15, 75 - 79, 1975.

Eb81 J. Ebert, "A Sensitive Transitive closure Algorithm," *Inf. Process Letters* 12, 5 (1981).

EK77 J. Eve and R. Kurki-Suonio, "On Computing the Transitive Closure of a Relation," *Acta Informatica* 8, 303 - 314, 1977.

HL82 R.L. Haskin and R.A. Lorie, "On Extending the Functions of a Relational Database System," *Proc. ACM SIGMOD Conf.*, Orlando, Fla., 1982, pp. 207-212.

IK83 T. Ibaraki and N. Katoh, On-line Computation of transitive closure for graphs, *Information Processing Letters*, 16:95-97, 1983.

It86 G.F. Italiano, Amortized effieciency of a path retrieval data structure, *Theorectical Computer Science*, 48:273-281, 1986.

IRW93 Y.E. Ioannidis, R. Ramakrishnan and L. Winger, "Tansitive Closure Algorithms Based on Depth-First Search," *ACM Trans. Database Syst.*, Vol. 18. No. 3, 1993, pp. 512 - 576.

Ja90 H.V. Jagadish, "A Compression Technique to Materilize Transitive Closure," *ACM Trans. Database Systems*, Vol. 15, No. 4, 1990, pp. 558 - 598.

KGM91 T. Keller, G. Graefe and D. Maier, "Efficient Assembly of Complex Objects," *Proc. ACM SIGMOD conf.* Denver, Colo., 1991, pp. 148-157.

Ki93 W. Kim, "Object-Oriented Database Systems: Promises, Reality, and Future," *Proc. 19th VLDB conf.*, Dublin, Ireland, 1993, pp. 676-687.

Kn73 D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Pub. London, 1973.

KR98 H.A. Kuno and E.A. Rundensteiner, "Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10. No. 5, 1998, pp. 768-792.

LL98 W.C. Lee and D.L Lee, "Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10. No. 3, 1998, pp. 371-388.

LMP87 B. Lindsay, J. McPherson and H. Pirahesh, "A Data Management Extension Architecture," *Proc. ACM SIGMOD conf.*, 1987, pp. 220-226.

Me84 K. Mehlhorn, "*Graph Algorithms and NP-Completeness*: *Data Structure and Algorithm 2*" Springer-Verlag, Berlin, 1984.

MMM97 A.O. Mendelzon, G.A. Mihaila, T. Milo, "Querying the World Wide Web," *Int. J. Digital Libraries*, Vol. 1, No. 1, April 1997, pp. 54 - 67.

Pu70 P. Purdom, "A Transitive Closure Algorithm," *BIT* 10, 76 - 94, 1970.

Ru95 R. Ramakrishnan and J.D. Ullman, "A Survey of Research in Deductive Database Systems," *J. Logic Programming*, May, 1995, pp. 125-149.

Sc83 L. Schmitz, "An Improved Transitive Closure Algorithm," *Computing* 30, 359 - 371 (1983).

SRH90 M. Stonebraker, L. Rowe and M. Hirohama, "The Implementation of POSTGRES," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 1, 1990, pp. 125-142.

Ta72 R. Tarjan: Depth-first Search and Linear Graph Algorithms, *SIAM J. Compt*. Vol. 1. No. 2. June 1972, pp. 146 -140.

Te96 J. Teuhola, "Path Signatures: A Way to Speed up Recursion in Relational Databases," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 3, June 1996, pp. 446 - 454.

Wa62 S. Warshall, "A Theorem on Boolean Matrices," *JACM*, 9. 1(Jan. 1962), 11 - 12.

Wa75 H.S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun. ACM* 18, 4 (April 1975), 218 - 220.

VB86 P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," in: *Proc. 1st Workshop on Expert Database Systems*, Charleston, S.C., 1986, pp. 197 - 208.

VKC86 P. Valduriez, S. Khoshafian and G. Copeland, "Implementation Techniques of Complex Objects," *Proc. 12th VLDB Conf.*, Kyoto, Japan, 1986, pp. 101-109.

Appendix A. Sample trace of graph decomposition

In this Appendix, we trace the algorithm *find-forest* against

the tree shown in Fig. 2(a).

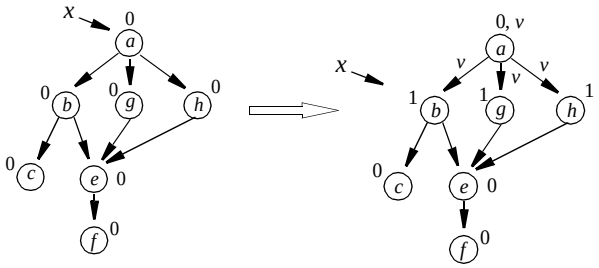

Fig. 6. The first execution step of *find-node-disjunct-forest*

See Fig. 6. At the beginning, every $l(u)$ is set to 0. After the first loop, the l-value of node a $l(a)$ remains 0. But the l-values of b, g, and h are changed to 1. Moreover, node a, and edge (a, b), (a, g) and (a, h) are marked with "v" to indicate that have been visited. In addition, part of E_1 has been generated. The rest steps are listed in Fig. 7, 8, 9 and 10.

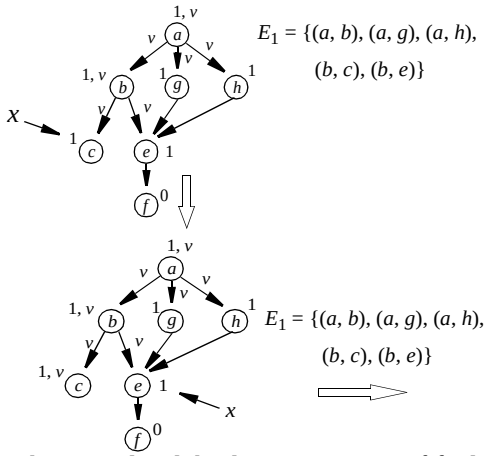

Fig. 7. The second and third execution step of *find-forest*

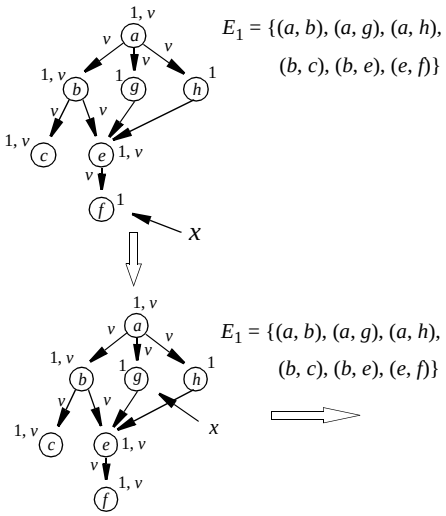

Fig. 8. The fourth and fifth execution step of *find-forest*

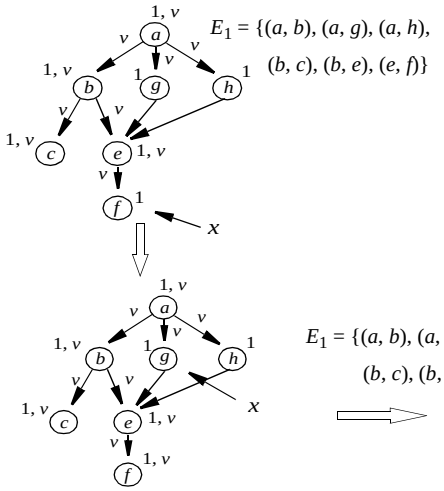

Fig. 9. The fourth and fifth execution step of *find-forest*

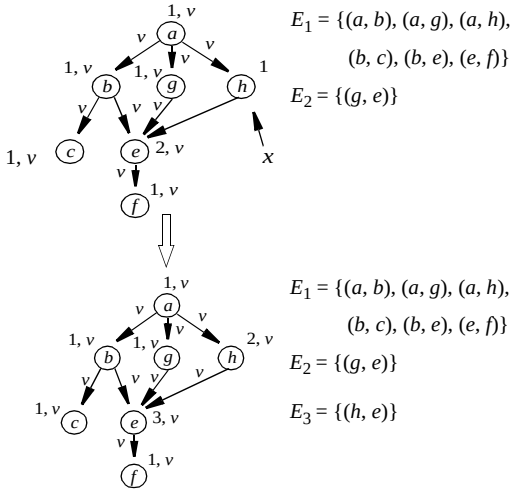

Fig. 10. The sixth and seventh execution step of *find-forest*