A Space-Economic Representation of Transitive Closures in Relational Databases

Yangjun Chen*

Dept. Applied Computer Science, University of Winnipeg,
515 Portage Ave. Winnipeg, Manitoba, Canada R3B 2E9
ychen2@uwinnipeg.ca

ABSTRACT

A composite object represented as a directed graph (digraph for short) is an important data structure that requires efficient support in *CAD/CAM, CASE, office systems, software management, web databases,* and *document databases.* It is cumbersome to handle such objects in relational database systems when they involve ancestor-descendant relationships (or say, recursive relationships). In this paper, we present a new encoding method to label a digraph, which reduces the footprints of all previous strategies. This method is based on a tree labeling method and the concept of *branchings* that are used in graph theory for finding the shortest connection networks. A branching is a subgraph of a given digraph that is in fact a forest, but covers all the nodes of the graph. On the one hand, the proposed encoding scheme achieves the smallest space requirements among all previously published strategies for recognizing recursive relationships. On the other hand, it leads to a new algorithm for computing transitive closures for *DAGs* (directed acyclic graph) in O($e \cdot b$) time and O($n \cdot b$) space, where n represents the number of the nodes of a DAG, e the numbers of the edges, and b the DAG's breadth. In addition, this method can be extended to cyclic digraphs and is especially suitable for a relational environment.

Keywords: Directed Graphs, Trees, Transitive Closures, Branchings, Graph Encoding

1. INTRODUCTION

It is a general opinion that relational database systems are inadequate for manipulating composite objects that arise in novel applications such as web and document databases [3, 12, 13, 35, 49], CAD/CAM, CASE, office systems and software management [8, 28, 43]. Especially, when recursive relationships are involved, it is cumbersome to handle them in relational databases, which sets current relational systems far behind the navigational ones [31].

A composite object can be generally represented as a directed graph (digraph). For example, in a CAD database, a composite object corresponds to a complex design, which is composed of several subdesigns [8]. Often, subdesigns are shared by more than one higher-level designs, and a set of design hierarchies thus forms a directed acyclic graph (DAG). As another example, the citation index of scientific literature, recording reference relationships between authors, constructs a directed cyclic graph. As a third example, we consider the traditional organization of a company, with a variable number of manager-subordinate levels, which can be represented as a tree hierarchy.

In a relational system, composite objects must be fragmented across many relations, requiring joins to gather all the parts. A typical approach to improving join efficiency is to equip relations with hidden pointer fields for coupling the tuples to be joined [11]. The so-called *join index* is another auxiliary access path to mitigate this difficulty [46, 47]]. Also, several advanced join algorithms have been suggested, based on hashing and a large main memory. In addition, a different kind of attempts to attain a compromise solution is to extend relational databases with new features, such as *clustering* of composite objects, by which the concatenated foreign keys of ancestor paths are stored in a primary key (see [23, 33, 39] for detailed description). Another extension to relational system is *nested relations* (or NF^2 relations, see, e.g., [18]). Although it can be used to represent composite objects without sacrificing the relational theory, it suffers from the problem that subrelations cannot be shared. Moreover, recursive relationships cannot be represented by simple nesting because the depth is not fixed. Finally, *deductive database*s and *object-relational databases* can be considered as two quite different extensions to handle this problem [29, 37, 15].

In this paper, we discuss a new encoding approach to pack "ancestor paths" in a relational environment. The main idea of this method is *tree labeling,* by means of which we associate each node v with a pair of integers (α, β) such that if v', another node associated with (α', β'), is a descendant of v, some arithmetical relationship between α and α', as well as β and β' can be determined. Then, such relationships can be used to find all descendants of a node, and the recursive closure w.r.t. a tree can be computed very efficiently. This method can be generalized to DAGs or digraphs containing cycles by decomposing a graph into a series of trees, for which the approach described above can be employed. As we can see later, a new method for computing recursion efficiently in a relational environment can be developed based on these techniques. In fact, it is a new algorithm to handle this problem with a representation of transitive closures different from traditional ones. It needs only O($e \cdot b$) time and O($n \cdot b$) space, where b is the breadth of the graph, defined to be the least number of disjoined paths that cover all the nodes of a graph. This computational complexity is better than any existing method for this problem, including the graph-based algorithms [20, 21, 34, 36, 38], the graph encoding [1, 2, 6, 7, 14, 16, 46, 46] and the matrix-based algorithms [24, 32, 44,

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

45]. (See Section 6 for a detailed comparison.)

The remainder of the paper is organized as follows. Section 2 specifies the problem to be solved. Section 3 introduces the concept of tree labeling, and applies it to simple tree-structured recursion hierarchies. Section 4 elaborates the technique to arbitrary recursion pattern. In Section 5, we address how to change labels when a new node or a new edge is inserted into a DAG. In Section 6, we discuss and compare the relevant work. Section 7 reports test results. Finally, a short conclusion is set forth in Section 8.

2. TASK DEFINITION

We consider composite objects represented by a digraph, where nodes stand for objects and edges for parent-child relationships, stored in a binary relation. In many applications, the transitive closure of a digraph needs to be computed, which is defined to be all ancestor-descendant pairs.

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. For example, the graph shown in Fig. 1(b) is the transitive closure of the graph shown in Fig. 1(a).

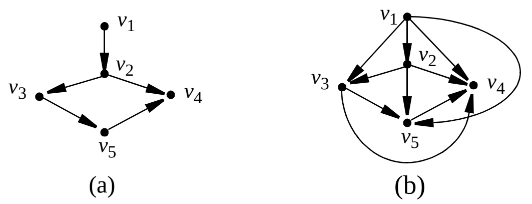


Fig. 1. A DAG and its transitive closure

A lot of research has been done on this issue. Among them, the semi-naive [10] and the logarithmic [46] are typical *algorithmic* solutions. Another main approach is the materialization of a transitive closure, either partially or completely [1, 2, 5, 6, 7, 49]. The implementation of the transitive closure algorithms in a relational environment has received extensive attention, including performance and the adaptation of the traditional algorithms [2, 4, 5, 6, 19, 26, 43].

The method proposed in this paper can be characterized as a partial materialization method. Given a node, we want to compute all its descendants efficiently based on a specialized data structure. The following is a typical structure to accommodate part-subpart relationships [17]:

- Part(<u>Part-id</u>, ...)
- Connection(<u>Parent-id, Child-id</u>, ...)

where Parent-id and Child-id are both foreign keys, referring to Part-id. In order to speed up the recursion evaluation, we'll associate each node with a pair of integers, which helps to recognize ancestor-descendant relationships.

In the rest of the paper, the following three types of digraphs will be discussed.

(i) Tree hierarchy, in which the parent-child relationship is of one-to-many type, i.e., each node has at most one parent.

(ii) Directed acyclic graph (DAG), which occurs when the relationship is of many-to-many type, with the restriction that a part cannot be sub/superpart of itself (directly or indirectly).

(iii) Directed cyclic graph, which contains cycles.

Later we'll use the term *graph* to refer to the *directed graph*, since we do not discuss non-directed ones at all.

3. TREE LABELING AND RECURSION COMPUTATION

In this section, we discuss how to label a tree to speed up the computation of recursion in a relational environment.

Consider a tree *T*. By traversing *T* in *preorder*, each node *v* will obtain a number (it can an integer or a real 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 relationships as follows.

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 Exercise 2.3.2-20 in [30]. □

If *v*' is a descendant of *v*, then we know that *pre*(*v'*) > *pre*(*v*) according to the preorder search. Now we assume that *post*(*v*') > *post*(*v*). Then, according to the postorder search, either *v*' is in some subtree on the right side of *v*, or *v* is in the subtree rooted at *v*', which contradicts the fact that *v*' is a descendant of *v*. Therefore, *post*(*v*') must be less than *post*(*v*). The following example helps for illustration.

Example 1. See the pairs associated with the nodes of the graph shown in Fig. 2. The first element of each pair is the preorder number of the corresponding node and the second is its postorder number. With such labels, the ancestor-descendant relationships can be easily checked.

For instance, by checking the label associated with b against the label for f, we see that b is an ancestor of f in terms of Proposition 1. Note that b's label is (2, 4) and f's label is (5, 2), and we have 2 < 5 and 4 > 2. We 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*. □

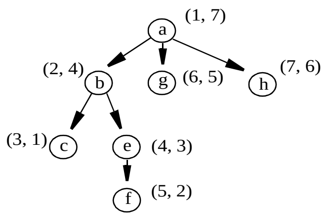


Fig. 2. Labeling a tree

Definition 1. (*label pair subsumption*) Let (*p*, *q*) and (*p*', *q*') be two pairs associated with nodes *u* and *v*. 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*').

To the best of our knowledge, this nice property has never been utilized in the database research or in the subtyping tests to recognize the ancestor-descendant relationships. Although in the *relative numbering* [40] the postorder numbers are used, it works in a quite different way. With this strategy, each node *v* is labeled with two numbers: the largest and the smallest postorder numbers in the subtree rooted at *v*. It is a more complicated labeling process than the tree labeling shown above. The *range-compression* [2] is a generalization of the relative numbering to DAGs. Its space requirement is on O(n^2) and it is different from our general strategy to be discussed in the next section. In [49], to speed up evaluation of containment queries, a node *v* (which represents an element in a document tree) is associated with a triple (*begin*, *end*, *level*), where *begin* is in fact a preorder number, *end* is the largest preorder number in the subtree rooted at *v*, and *level* is the depth, at which *v* is located. Therefore, it is also quite different from our method.

According to the tree labeling discussed above, the relational schema to handle recursion can consists of only one relation of the following form:

Node(Node_id, *label_pair*, ...),

where label_pair is used to accommodate the preorder and the postorder numbers of the nodes of a graph, denoted label_pair.*preorder* and label_pair.*postorder*, respectively. Then, to retrieve the descendants of node *x*, we issue two queries as below.

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

Let the label pair obtained by evaluating Q_1 be *y*. Then, the second query is of the following form:

```
Q2:   SELECT   *
      FROM     Node
      WHERE    label_pair.preorder > y.preorder
               and label_pair.postorder < y.postorder
```

From this, we can see that with the tree labeling, the recursion w.r.t. a tree can be handled conveniently in a relational environment.

4. GENERALIZATION

Now we discuss how to recognize the ancestor-descendant relationships w.r.t. a general structure: a DAG or a graph containing cycles. First, we address the problem of DAGs in 4.1. Then, cyclic graphs will be discussed in 4.2. 4.3 is devoted to the computation of transitive closures.

4.1 Recursion w.r.t. DAGs

What we want is to apply the technique discussed above to a DAG. To this end, we establish a *branching* of the DAG as follows.

Definition 2. (*branching* [T42]) A subgraph $B = (V, E')$ of a digraph $G = (V, E)$ is called a branching if it is cycle-free and $d_{indegree}(v) \leq 1$ for every $v \in V$. □

Clearly, if for only one node *r*, $d_{indegree}(r) = 0$, and for all the rest of the nodes, *v*, $d_{indegree}(v) = 1$, then the branching is a directed tree with root *r*. Normally, a branching is a set of directed trees. Now, we assign every edge *e* a same cost (e.g., let cost $c(e) = 1$ for every edge *e*). We will find a branching for which the sum of the edge costs, $\sum_{e \in E'} c(e)$, is maximum.

For example, the trees shown in Fig. 3(b) are a maximal branching of the graph shown in Fig. 3(a) if each edge has a same cost. Assume that the maximal branching for $G = (V, E)$ is a set of trees T_i with root r_i ($i = 1, ..., m$). We introduce a *virtual root r* for the branching and an edge $r \rightarrow r_i$ for each T_i, obtaining a tree G_r, called the representation of *G*. For instance, the tree shown in Fig. 3(c) is the representation of the graph shown in Fig. 3(a). Using Tarjan's algorithm for finding optimum branchings [42], we can always find a maximal branching for a directed graph in O(|*E*|) time if the cost for every edge is equal to each other. Therefore, the representative tree for a DAG can be constructed in linear time.

By traversing G_r in *preorder*, each node *v* will obtain a number *pre*(*v*); and by traversing G_r in *postorder*, each node *v* will get another number *post*(*v*). These two numbers can be used to recognize the ancestor-descendant relationships of all G_r's nodes as discussed in Section 3.

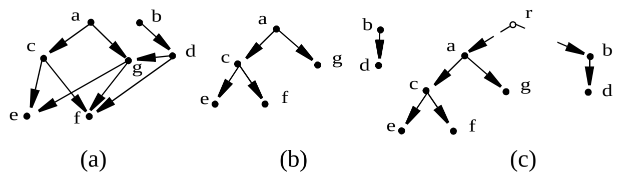


Fig. 3. A DAG and its branching

In a G_r (for some *G*), a node *v* can be considered as a representation of the subtree rooted at *v*, denoted $T_{sub}(v)$; and the pair (*pre*, *post*) associated with *v* can be considered as a pointer to *v*, and thus to $T_{sub}(v)$. (In practice, we can associate a pointer with such a pair to point to the corresponding node in G_r.) In the following, what we want is to construct a pair sequence: (pre_1, $post_1$), ..., (pre_k, $post_k$) for each node *v* in *G*, representing the union of the subtrees (in G_r) rooted respectively at (pre_j, $post_j$) (j = 1, ..., *k*), which contains all the descendants of *v*. In this way, the space overhead for storing the descendants of a node is dramatically reduced. Later we will shown that a pair sequence contains at most O(*b*) pairs, where *b* is the breadth of *G*. (The breadth of a digraph is defined to be the least number of the disjoint paths that cover all the nodes of the graph.)

Example 2. The representative tree G_r of the DAG *G* shown in Fig. 3(a) can be labeled as shown in Fig. 4(a). Then, each of the generated pairs can be considered as a representation of some subtree in G_r. For instance, pair (3, 3) represents the subtree rooted at *c* in Fig. 4(a).

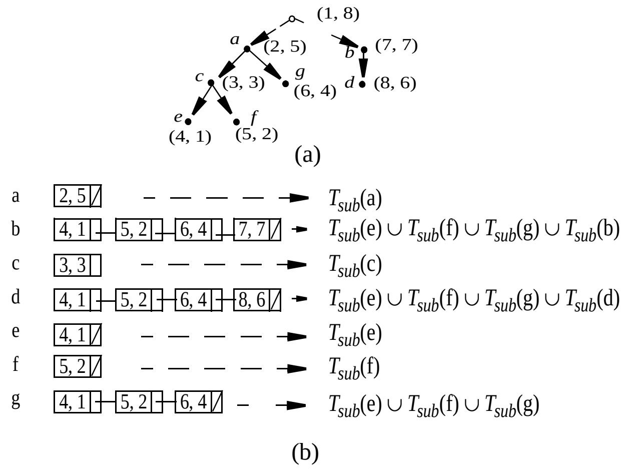

(1, 8)
a (2, 5)
b (7, 7)
c (3, 3)
g (6, 4)
d (8, 6)
e (4, 1)
f (5, 2)
(a)
a 2, 5 → T_{sub}(a)
b 4, 1 5, 2 6, 4 7, 7 → T_{sub}(e) ∪ T_{sub}(f) ∪ T_{sub}(g) ∪ T_{sub}(b)
c 3, 3 → T_{sub}(c)
d 4, 1 5, 2 6, 4 8, 6 → T_{sub}(e) ∪ T_{sub}(f) ∪ T_{sub}(g) ∪ T_{sub}(d)
e 4, 1 → T_{sub}(e)
f 5, 2 → T_{sub}(f)
g 4, 1 5, 2 6, 4 → T_{sub}(e) ∪ T_{sub}(f) ∪ T_{sub}(g)
(b)

Fig. 4. Tree labeling and illustration for transitive closure representation

If we can construct, for each node v, a pair sequence as shown in Fig. 4(b), where it is stored as a linked list, the descendants of the nodes can be represented in an economical way. Let $L = (pre_1, post_1), ..., (pre_k, post_k)$ be a pair sequence and each $(pre_i, post_i)$ is a pair labeling v_i ($i = 1, ..., k$). Then, L corresponds to the union of the subtrees $T_{sub}(v_1)$, ..., and $T_{sub}(v_k)$. For example, the pair sequence (4, 1)(5, 2)(6, 4)(8, 6) associated with d in Fig. 4(b) represents a union of 4 subtrees: T_{sub}(e), T_{sub}(f), T_{sub}(g) and T_{sub}(d), which contains all the descendants of d in G. □

The question is how to construct such a pair sequence for each node v so that it corresponds to a union of some subtrees in G_r, which contains all the descendants of v in G.

First, we notice that by labeling G_r, each node in $G = (V, E)$ will be initially associated with a pair as illustrated in Fig. 5. That is, if a node v is labeled with (*pre*, *post*) in G_r, it will be initially labeled with the same pair (*pre*, *post*) in G.

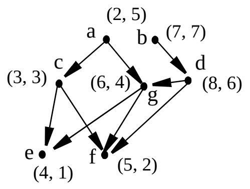


Fig. 5. Graph labeling

To compute the pair sequence for each node, we sort the nodes of G topologically, i.e., $(v_i, v_j) \in E$ implies that v_j appears before v_i in the sequence of the nodes. The pairs to be generated for a node v are simply stored in a linked list A_v. Initially, each A_v contains only one pair produced by labeling G_r.

We scan the topological sequence of the nodes from the beginning to the end and at each step we do the following:

Let v be the node being considered. Let $v_1, ..., v_k$ be the children of v. Merge A_v with each A_{v_l} for the child node v_l ($l = 1, ..., k$) as follows. Assume $A_v = p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_g$ and $A_{v_l} = q_1 \rightarrow q_2 \rightarrow ... \rightarrow q_h$, as shown in Fig. 6. Assume that both A_v and A_{v_l} are increasingly ordered. (We say a pair p is larger than another pair p', denoted $p > p'$ if $p.pre > p'.pre$ and $p.post > p'.post$.)

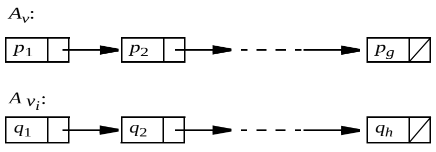


Fig. 6. linked lists associated with nodes in G

We step through both A_v and A_{v_l} from left to right. Let p_i and q_j be the pairs encountered. We'll make the following checkings.

(1) If $p_i.pre > q_j.pre$ and $p_i.post > q_j.post$, insert q_j into A_v after p_{i-1} and before p_i and move to q_{j+1}.
(2) If $p_i.pre > q_j.pre$ and $p_i.post < q_j.post$, remove p_i from A_v and move to p_{i+1}. (*p_i is subsumed by q_j.*)
(3) If $p_i.pre < q_j.pre$ and $p_i.post > q_j.post$, ignore q_j and move to q_{j+1}. (*q_j is subsumed by p_i; but it should not be removed from A_{v_l}.*)
(4) If $p_i.pre < q_j.pre$ and $p_i.post < q_j.post$, ignore p_i and move to p_{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 linked 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   $p \leftarrow first\text{-}element(A_1)$;
2   $q \leftarrow first\text{-}element(A_2)$;
3   while $p \neq nil$ do{
4     while $q \neq nil$ do{
5       if ($p.pre > q.pre \wedge p.post > q.post$) then
6          {insert $q$ into $A_1$ before $p$;
7            $q \leftarrow next(q)$;}
                  (*$next(q)$ represents the pair next to $q$ in $A_2$.*)
8       else if ($p.pre > q.pre \wedge p.post < q.post$) then
9            {$p' \leftarrow p$; (*$p$ is subsumed by $q$; remove $p$ from $A_1$.*)
10            remove $p$ from $A_1$;
11            $p \leftarrow next(p')$;}
                  (*$next(p')$ represents the pair next to $p'$ in $A_1$.*)
12           else if ($p.pre < q.pre \wedge p.post > q.post$) then
13                 {$q \leftarrow next(q)$;}
          (*$q$ is subsumed by $p$; move to the next element of $q$.*)
14                 else if ($p.pre < q.pre \wedge p.post < q.post$) then
15                      {$p \leftarrow next(p)$;}
16                      else if ($p.pre = q.pre \wedge p.post = q.post$)
17                           then {$p \leftarrow next(p)$; $q \leftarrow next(q)$;}
18  if $p = nil \wedge q \neq nil$ then {attach the rest of $A_2$ to the end of $A_1$;}
end
```

The following example helps for illustration.

Example 3. Assume that A_1 = (7, 7)(11, 8) and A_2 = (4, 3)(8, 5)(10, 11). Then, A = *pair-sequence-merge*(A_1, A_2) = (4, 3)(7, 7)(10, 11). Fig. 7 shows the entire merging process.

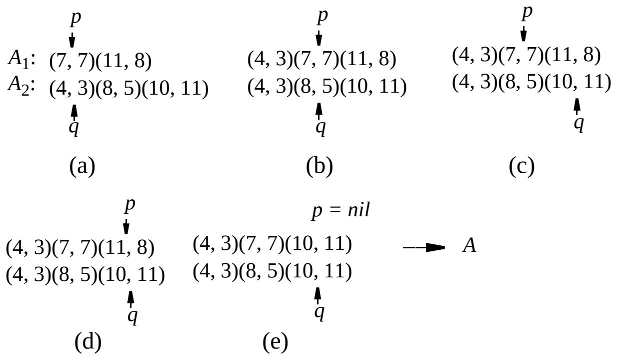


Fig. 7. An entire merging process

In each step, the A_1-pair pointed by p and the A_2-pair pointed by q

are compared. In the first step, (7, 7) in A_1 will be checked against (4, 3) in A_2 (see Fig. 7(a)). Since (4, 3) is smaller than (7, 7), it will be inserted into A_1 before (7, 7) (see Fig. 7(b)). In the second step, (7, 7) in A_1 will be checked against (8, 5) in A_2. Since (8, 5) is subsumed by (7, 7), we move to (10, 11) in A_2 (see Fig. 7(c)). In the third step, (7, 7) is smaller than (10, 11) and we move to (11, 8) in A_1 (see Fig. 7(d)). In the fourth step, (11, 8) in A_1 is checked against (10, 11) in A_2. Since (11, 8) is subsumed by (10, 11), it will be removed from A_1 and p becomes *nil* (see Fig. 7(e)). In this case, (10, 11) will be attached to A_1 (see line 18 of Algorithm *pair-sequence-merge*( )), forming the result A = (4, 3)(7, 7)(10, 11) (see Fig. 7(e)). Fig. 8 is a pictorial illustration of the result of merging A_2 into A_1.

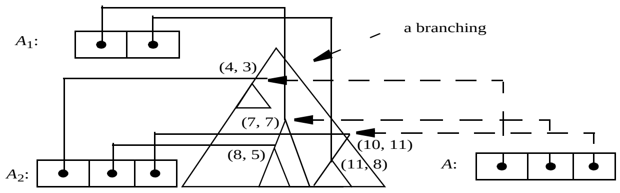


Fig. 8. Illustration of merging two pair sequences

From this, we can see that A_1 represents the union of two subtrees respectively rooted at (7, 7) and (10, 11) while A_2 represents the union of three subtrees rooted at (4, 3), (8, 3) and (10, 11), respectively. The result A got by merging A_2 into A_1 is the union of three subtrees rooted at (4, 3), (7, 7) and (10, 14), respectively (see the dashed arrows in Fig. 7). □

In the following, we establish several propositions to clarify the properties of the above algorithm.

Proposition 2. 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*( ). Then, A is also sorted increasingly.

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. Obviously, removing a pair from A_1 will not change the ordering of A_1. Let q be a pair of A_2 inserted into A_1. It may be done at line 6 or at line 18. If it is done at 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 into A_1 at all. In this case, the proposition holds. If q is inserted into A_1 by executing line 18, all the pairs in A_1 must be used up before the line 18 is carried out. 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 originally in increasing order. Therefore, in this case, the proposition holds, too. □

Proposition 3. 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 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_1, then there must be a pair in A such that the subtree rooted at it contains v.

Proof. Similar to Proposition 3. □

Proposition 5. The time complexity of Algorithm *pair-sequence-merge* is bounded by $O(|A_1| + |A_2|)$.

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 discussed above, the pair sequences for all the nodes in a DAG can be computed as follows.

```
Algorithm all-sequence-generation
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($A_i$, $A_{i_j}$);
6       }
end
```

Proposition 6. The space complexity of Algorithm *all-sequence-generation* is bounded by $O(n \cdot b)$, where b is the breadth of G and n is the number of the nodes of G.

Proof. In the algorithm, each node v is associated with a linked list A_v. We claim that the size of A_v is bounded by b. Assume that A_v contains $b + 1$ pairs that are different from each other. Then, there must exist two pairs p and q so that p subsumes q or *vice versa*. Therefore, one of them will be removed. Thus, the space needed for Algorithm *all-sequence-generation* is bounded by $O(n \cdot b)$. □

Proposition 7. The time complexity of Algorithm *all-sequence-generation* is bounded by $O(e \cdot b)$, where b is the breadth of G and e is the number of the edges of G.

Proof. In the out for-loop of the algorithm, n steps are performed. In each step, d_i merge operations are made for each v_i, where d_i represents the outdegree of v_i. Therefore, the time spent for each step is $O(d_i \cdot b)$. The whole time complexity is thus $O(\sum d_i \cdot b) = O(e \cdot b)$. □

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 $O(e \cdot b)$ according to Proposition 7, and the space complexity is bounded by $O(n \cdot b)$ according to Proposition 6. This computational complexity is superior to any existing ones. We compare our algorithm with the relevant work in Section 6.

Proposition 8. Let v be a node in G. Any descendant u of v must be in a subtree of G_r rooted at a node labeled with a pair in A_v constructed by Algorithm *all-sequence-generation*.

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 l in the topological sequence.

Basis. When $l = 1$, v_n is a leaf node in G and its linked 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 linked list A_i associated with v_i ($i = n, ..., n - k$) contains all the pairs covering all the descendants of v_i.

Consider $l = k + 1$. 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 *all-sequence-generation*, as well as Proposition 2, 3 and 4, we can see that the linked 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. □

Example 4. A possible topological sequence for the graph shown in Fig. 3(a) is: e → f → g → c → d → a → b. The pairs associated with them are: (4, 1), (5, 2), (6, 4), (3, 3), (8, 6), (2, 5) and (7, 7), respectively. They are obtained by labeling the tree shown in Fig. 4(a). Applying Algorithm *all-sequence-generation* to this sequence, we will produce a linked list for each of them as shown in Fig. 9. □

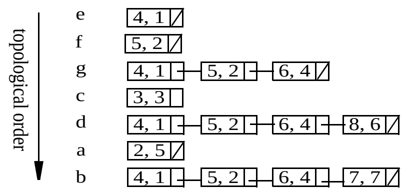


Fig. 9. linked lists representing pair sequences

We can store physically the label pair for each node, as well as its label pair sequence produced using Algorithm *all-sequence-generation*. Concretely, the relational schema to handle recursion w.r.t. a DAG can be established in the following form:

Node(Node_id, *label*, *label_sequence*, ...),

where *label* and *label_sequence* are used to accommodate the label pairs and the label pair sequences associated with the nodes of a graph, respectively. Then, to retrieve the descendants of node x, we issue two queries. The first query is similar to Q_1:

```
Q3:    SELECT    label_sequence
       FROM      Node
       WHERE     Node_id = x.
```

Let the label sequence obtained by evaluating Q_3 be y. Then, the second query will be of the following form:

```
Q4:    SELECT    *
       FROM      Node
       WHERE     φ(label, y),
```

where $\phi(p, s)$ is a boolean function with the input: p and s, where p is a pair and s a pair sequence. If there exists a pair p' in s such that $p \prec p'$ (i.e., $p.pre > p'.pre$ and $p.post < p'.post$), then $\phi(p, s)$ returns *true*; otherwise *false*.

We note that each execution of ϕ function needs only $O(\log|s|)$ time. This can be done as follows. Assume $s = p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_g$, which is stored in two arrays A_v^1 and A_v^2 : one for all the $p_i{\cdot}pre$s and the other for all the $p_i{\cdot}post$s. To check whether p is subsumed by a p_i, we make a binary search of A_v^1 to find the largest i such that $p_i{\cdot}pre < p{\cdot}pre$. Then, we search A_v^2 from the 1st to the ith element to find whether there exists j such that $p_j{\cdot}post < p{\cdot}post$. If it is the case, p is subsumed by some pair in s; otherwise not. Obviously, the time complexity of this process is $O(\log|s|)$. Since $|s|$ is bounded by the breadth b of the graph, this method requires only $O(n \cdot \log|b|)$ time to find all the descendants of a given node.

4.2 Recursion w.r.t. cyclic graphs

Based on the method discussed in the previous subsection, we can easily develop an algorithm to compute recursion for cyclic graphs. First, we use Tarjan's algorithm for identifying *strongly connected components* (*SCCs*) to find the cycles of a cyclic graph [41] (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. Next, we handle the DAG as discussed in 4.1. In this way, however, all nodes in an SCC will be assigned the same pair (and the same pair sequence). For this reason, the method for computing the recursion at some node x should be slightly changed. In the following, Q_5 is the same as Q_3, but Q_6 is different from Q_4.

```
Q5:    SELECT    label_sequence
       FROM      Node
       WHERE     Node_id = x
```

Let the label sequence obtained by evaluating Q_5 be y. Then, the second query will be of the following form:

```
Q6:    SELECT    *
       FROM      Node
       WHERE     χ(label, y),
```

where $\chi(t, s)$ is a boolean function with the input: t and s, where t is a pair and s a pair sequence. If there exists a pair t' in s such that $t'.pre \leq t.pre$ and $t'.post \geq t.post$, then $\chi(t, s)$ returns *true*; otherwise *false*.

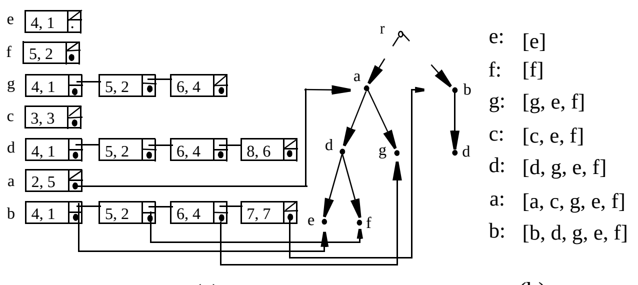
We note that $\chi(\ )$ is slightly different from $\phi(\ )$ defined in 4.1.

In Q_6, any two nodes in the same SCC are considered to be the descendants of each other.

Finally, we point out that the time complexity of computing recursion w.r.t. a cyclic graph is still $O(e \cdot b)$ since Tarjan's algorithm runs in $O(n + e)$ time [41].

4.3 About computation of transitive closures

The algorithm discussed in 4.1 is in essence a new strategy with a representation of transitive closures quite different from the traditional methods. As mentioned in 4.1, each pair in a pair sequence associated with a node in G can be considered as a pointer to a subtree in G_r. (In practice, each pair can be associated with a physical pointer as illustrated in Fig. 10(a).) Then, the union of all the subtrees pointed to by the pairs in a pair sequence associated with a node v contains all the descendants of v.



(a) (b)

Fig. 10. linked lists representing pair sequences

For an intuitive explanation, see the pair (2, 5) associated with a in Fig. 10(a). It is a pointer to the node a in the branching, indicating that all the nodes in the subtree rooted at a (in the branching) are the descendants of a in the original graph. As another example, see the pair sequence for b that contains 4 pairs, representing 4 pointers as shown in the figure. From the original graph shown in Fig. 3(a), we see that the union of the subtrees pointed to by these 4 pointers contains really all the descendants of b. In this way, we need only $O(n \cdot b)$ space to store a transitive closure, where b is the breadth of the graph. As discussed in 4.1, the time spent to generate the pair sequences for all the nodes in a graph is $O(e \cdot b)$.

In a traditional method, however, the transitive closure of a graph is represented by a set of lists and each list contains all the descendants of some node (see Fig. 10(b)) or by a $n \times n$ boolean matrix Therefore, the space requirement for storing a transitive closure is $O(n^2)$.

5. INCREMENTAL CHANGES TO GRAPHS

In a dynamic environment, the data in a database can be frequently changed. With a static encoding, the recomputation of labels is needed, which is normally very time consuming. However, in our method, adding a node to a DAG needs only $O(b)$ time for calculating its label. Adding an edge is more difficult since it may cause $O(n^2)$ edges in the transitive closure of a graph (see [27]). In our method, it needs $O(b)$ time in the best case and $O(n \cdot b)$ time in the worst case. But for n consecutive insertions of edges, it takes only $O(n \cdot b)$ time. Therefore, the average time for inserting an edge is $O(b)$. In the following, we discuss the node insertion and the edge insertion in 5.1 and 5.2, respectively.

5.1 Node insertion

We notice that a label can be a real number, not always an integer. Between two consecutive integers, we have infinitely many real numbers. Then, when a new node is inserted into a graph, we can assign a real number to a node as its preorder or postorder number, which will not twist the mechanism discussed in the preceding section. However, infinitely many numbers are not implementable. Depending on different applications, we set different numbers of digits to the right of the decimal point for the possible insertion. It can be considered as an extension to the idea discussed in [2], which leaves gaps between each pair of successive postnumbers for inserting nodes.

In the following, we first discuss the node insertion into a tree. Then, the node insertion into a DAG is briefly outlined.

- Node insertion into a tree

First, we consider two cases of node insertion into a tree.

1. A new node v is inserted into a tree T as a direct right sibling of some node u.
2. A new node v is inserted into a tree T as a parent of some node u.

In these two cases, the pair (*pre*, *post*) associated with v is calculated as follows.

Let the pair associated with u be (p, q). Let the pair associated with the node s preceding u (according to the preorder numbering) be (p_s, q_s). Let the pair associated with the node t next to s (according to the postorder numbering) be (p_t, q_t). We have

$$pre = p_s + \frac{p - p_s}{2}\text{, and } post = q_t - \frac{q_t - q}{2}\,.$$

Obviously, if each node keeps a pointer to its predecessor (according to the preorder numbering) and a pointer to its successor (according to the postorder numbering), this operation needs only a constant time.

Now we consider another two cases that are dual to case 1 and 2, respectively.

3. A new node v is inserted into a tree T as a direct left sibling of some node u.
4. A new node v is inserted into a tree T as a child of some node u and the parent of one of u's children.

Let the pair associated with u be (p, q). Let the pair associated with the node s preceding u (according to the postorder numbering) be (p_s, q_s). Let the pair associated with the node t next to s (according to the preorder numbering) be (p_t, q_t). We have

$$pre = p + \frac{p_t - p}{2}\text{, and } post = q - \frac{q - q_s}{2}\,.$$

- Node insertion into a DAG

To calculate the pair and the pair sequence for a new node inserted into a DAG, we have to determine where to insert the node in the corresponding branching. This can be done by checking the parent of the new node, which is an easy task since the information on its parent must be specified. Two cases will be considered:

1. The node is inserted as a child of some node in the branching.
2. The node is inserted between an edge (u, v) (i.e., it is inserted as a child of u and the parent of v.)

For both cases, we compute the pair for the new node using the method discussed above; but yet we need to merge the pair sequence of its child with its pair for case 2. The time complexity is O(b).

5.2 Edge insertion

Now we consider the insertion of an edge e into a DAG G. Let $e = (u, v)$. Let A_u and A_v be the label sequences for u and v, respectively. The following algorithm changes the labels of G when e is inserted into it.

```
Algorithm inserting-one-edge(G, e)
begin
  call pair-sequence-merge($A_v$, $A_u$);(*pair-sequence-merge($A_v$,
    $A_u$) was discussed in Section 3.*)
  if $A_v$ is not changed then return;
  else {Let $v_n$, $v_{n-1}$, ..., $v_1$ be the topological sequence of the
        nodes of G;
     Assume $v = v_l$ for some $l$;
     for i from l downto 1 do(*change the labels of the nodes
     along the topological order.*)
        {let $v_{i_1}$, ..., $v_{i_k}$ be the child nodes of $v_i$;
           for j from 1 to k do
*             if $A_{i_j}$ is not changed then call pair-sequence-
                 merge($A_i$, $A_{i_j}$);
**            if there are b consecutive label sequences not changed
                 (*b represents the breadth of G.*)
              then return;
           }
end
```

This algorithm is similar to the Algorithm *all-sequence-generation*. The main difference is the line marked with '*', where whether A_{i_j} is changed is checked before *pair-sequence-merge*(A_i, A_{i_j}) is invoked. Another difference is the line marked with '**', where we check whether there are b consecutive label sequences not changed (b represents the breadth of G). If it is the case, the algorithm terminates. Recall that the time complexity of Algorithm *pair-sequence-merge*( ) is O(b). Therefore, we need only O(b) time for inserting an edge in the case that A_v is not changed after A_u is merged into it. In the worst case, we will execute the algorithm *pair-sequence-merge*( ) l times (if v is the lth node in the topological sequence of G). For each of these nodes, we will merge all those changed pair sequences of its children with its pair sequence. (See the line marked '*' in the above algorithm.) Since at most n pair sequences may be changed, the time complexity is bounded by O($n \cdot b$).

We notice that inserting m ($m > 1$) edges consecutively into a DAG G does not require more time than inserting only one edge. This is because after changing the label sequences for all the nodes involved, we need to scan the topological sequence of G only once. Therefore, the time complexity of inserting m edges into G is bounded by O($m \cdot \frac{n}{\min\{m, n\}} \cdot b$).

In particular, if $m = n$, the time complexity is O($n \cdot b$).

6. RELEVANT WORK AND COMPARISON

In the past several decades, the transitive closure of a graph has be extensively researched and a lot of different strategies have been proposed, such as the graph-based algorithms [20, 21, 34, 36, 38], the graph encoding [1, 2, 7, 14, 16, 27, 43, 48] and the matrix-based methods [24, 32, 44, 45]. In the following, we mainly discuss the graph encoding and the matrix-based methods since they uniformly outperform the graph-based methods.

- *Graph encoding*

Perhaps the most elegant algorithm for encoding is *relative numbering* [40] (also called *Schubert's numbering*), by means of which each node v is associated with an interval $[q_v, p_v]$ to facilitate the checking of ancestor-descendant relationships, where p_v is the postorder number of v and q_v is the least postorder number of the subtree rooted at v. This method was generalized to DAGs by Agrawal, Borgida and Jagadish [2] into what is called *range-compression*. In this method, each node v is associated with a postorder number p_v and two arrays A_v and B_v such that if there exists another node u satisfying the following condition for some i: $A_u[i] < p_v < B_u[i]$, v is a descendant of u. Since the size of A_v (or B_v) is on the order O(n), the space overhead of this encoding is O(n^2). To generate such arrays, all the edges have to be accessed. Thus, the time overhead of this method is O($n \cdot e$). In addition, by leaving gaps between each pair of successive postorder numbers, inserting a node can be done efficiently. However, for a new node, its arrays has to be created and therefore, at least, O(n) time is needed. To insert an edge is more difficult since the data structures for all the descendants have to be recomputed, which needs in the worst case O(n^2) time. Finally, O(n) time is needed for path checking since both the arrays associated with a node will be scanned.

Bommel and Beck [7] improves the method proposed in [2] only by setting suitable gaps. Therefore, their method has the same computational complexities as [2].

In [27], Jagdish discussed a different graph encoding, by means of which, for any pair of nodes u and v with labels (i, j) and (k, j), if $i < k$, there exists a directed path from u to v. Since for all the nodes on each path (called a *chain* in [27]) such labels are established, the space overhead is O($c \cdot n$), where c is the number of paths covering all the nodes of a graph. Since the paths may not be disjoined (i.e., a node may appear on more than one path), the number of the labels associated with a node is on the order O(n) in the worst case. The time complexity of this method is O($n \cdot e$) and it needs O($c \cdot n$) time to insert an edge. Finally, inserting a node needs at least O(n) time since for each path where the node appears the labels have to be created, and the labels for all the descendants as well as all the ancestors have to be changed.

In the method proposed by Abdeddaim [1], a node v is associated with two integer sequences of the same length: $p = p_1, ..., p_k$ and $s = s_1, ..., s_k$, where k is the number of the disjoined paths covering the graph. Each p_i in p is the number of the nodes on a path P_i, which are the predecessors of v while each s_j in s is equal to $|P_j| - succ_j(v) + 1$, where $|P_j|$ is the number of nodes on P_j and $succ_j(v)$ is the number of the successors of v on P_j. With such sequences associated with the nodes, the path checking can be done in O(k) time. However, this method maintains the transitive closure of a graph in $O(k^2 \cdot e + n \cdot \min\{n, e\})$ time and $O(k \cdot n)$ space; and needs $O(n^2)$ time to modify relevant $succ_j(v)$'s when inserting an edge.

In [43], Teuhola discussed an interesting bit-sequence encoding. In this method, each node v is associated with an interval (l, h), where l and h are two signatures each consisting of a bit string. These bit strings are constructed in such a way that if the interval associated with a descendant of v is (l', h'), then $l \le l'$ *and* $h \ge h'$ hold. In the case of DAGs, the space and time overhead of this method are bounded respectively by $O(d \cdot n)$ and $O(d \cdot e)$, where d is the length of a signature. However, the length of a signature is sensitive to the height of a graph (which is defined to be the longest path in a graph) and the outdegrees of the nodes. Therefore, in the worst case, inserting a node or an edge needs to change all the signatures of the graph, leading to an $O(d \cdot e)$ time complexity. Obviously, the path checking only needs $O(d)$ time. The main problem of this method is that the graph decomposition is not clearly defined, which is needed to label a graph using signatures. Besides the above methods, several other encodings have been developed in the context of programming languages to speed up type-subtype checkings, such as *Cohen's encoding* [16], *Packed-Encoding* (also called *PE-Encoding*) [14] and *PQ-Encoding* [48]. Cohen's encoding can be used only for tree structures; and PE-Encoding is a bit-vector based strategy. The length of a bit-vector is on $O(n)$, where n is the number of the types (subtypes) in a hierarchy. The PE-Encoding was proposed by Zibin and Gil in 2001 [48], which associates each node v with an array of integers A_v and an interval $[l_v, h_v]$ such that for any ancestor u of v we have $l_u < A_v[i] < h_u$ for some i. The length of the array is shortened by decomposing a graph into slices in such a way that there is only an entry in the array for each slice. This encoding is based on the concept of PQ-trees, a data structure proposed in [9], which is used to test for the consecutive 1's property in binary matrices. However, since the size of a slice is bounded by a constant, the length of the array is on $O(n/g)$, where g represents average number of nodes in a slice. Therefore, the space overhead of the PQ-Encoding is on $O(n^2/g)$. The time overhead should be $O(n \cdot e/g)$. Inserting an edge and inserting a node are difficult since in the worst case, the slices have to be redefined, leading to a time complexity of $O(n \cdot e/g)$. In terms of the analysis of [48], a path checking needs $O(\log n)$ time.

From the discussion in Section 3, we can see that our algorithm is a new graph encoding different from any one outlined above and needs only $O(e \cdot b)$ time and $O(n \cdot b)$ space to compute the transitive closure of a graph. The time for Inserting a node is bounded by $O(b)$. In addition, we need only $O(\log b)$ time for path checking and $O(m \cdot \frac{n}{\min\{m, n\}} \cdot b)$ time for inserting m edges successively.

Table 1 shows the computational complexities of all the main graph encoding algorithms.

Table 1. Comparison of computation complexities.

- *Matrix-based methods*

In the computing graph theory, the transitive closure has been long an interesting topic. Many methods have been proposed based on the matrix representation, such as the methods discussed in [24, 32, 44, 43]. In such methods, a transitive closure is maintained as a matrix M with $M[i, j] = 1$ indicating that there exists a path from node i to node j in G. Among all these algorithms, the most interesting is Warren's [45]. It improves the time complexity of Washall's [45] from $O(n^3)$ to $O(n \cdot e)$. Since the transitive closure of a graph is stored in a matrix, the time for path checking is bounded by a constant. However, the space overhead is $O(n^2)$. In addition, even applying the well-known set-union algorithm, inserting m edges successively needs $O(m\alpha(m + e, n) + e + n)$ time, where $\alpha(x, y)$ is a very slowly growing function. Adding a node is also difficult since a $n \times n$ matrix will be changed to a $(n + 1) \times (n + 1)$ one. Therefore, it needs at least $O(n)$ time.

	Agrawal Borgida Jagadish	*Bommel Beck*	*Jagdish*	*Abdeddam*	*Teuhola*	*Zibin Gil*	*our graph labeling*
time	$O(n \cdot e)$	$O(n \cdot e)$	$O(n \cdot e)$	$O(k^2 \cdot e + n \cdot \min\{n, e\})$	$O(d \cdot e)$	$O(n \cdot e/g)$	$O(b \cdot e)$
space	$O(n^2)$	$O(n^2)$	$O(c \cdot n)$	$O(k \cdot n)$	$O(d \cdot n)$	$O(n^2/g)$	$O(b \cdot n)$
path check	$O(n)$	$O(n)$	$O(n)$	$O(k)$	$O(d)$	$O(\log n)$	$O(\log b)$
insert an edge	$O(n^2)$	$O(n^2)$	$O(c \cdot n)$	$O(n^2)$	$O(d \cdot e)$	$O(n \cdot e/g)$	$O((n/\min\{m, n\}) \cdot b)$
insert a node	$O(n)$	$O(n)$	$O(n)$	$\ge O(n)$	$O(d \cdot e)$	$O(n \cdot e/g)$	$O(b)$

The method proposed by Ibaraki and Katoh [24] maintains the transitive closure of a graph in the same computation complexity as Warren's. However, it needs only $O(n^3)$ time to insert q edges successively by using a special bit operation. When the order of m is larger, $O(n^3)$ is better than $O(m\alpha(m + e, n) + e + n)$.

In Italiano's method [25], an $n \times n$ matrix of pointers is used. Each of them points to a spanning tree. Therefore, more time and space than Warren's are needed. The goal of Italiano's was to get a better amortized time complexity for two operations: inserting an edge and checking a path (if such a path exists, return it). Using Italiano's data structure, each of the above two operations can be done in $O(n)$ amortized time.

The method discussed in [32] is a more complicated strategy. Instead of computing the transitive closure of a $G = (V, E)$, this method computes the transitive closure of G's reduction $G^- = (V, E^-)$, where E^- is defined as follows:

$E^- = E / \{(u, v) \mid (u, v) \mid (u, v) \in E$ and there is path P from u to v with $|P| \ge 2\}$.

Obviously, $|E^-| \le |E|$. Therefore, the computation of the transitive closure of G^- needs less time than G. But the time complexity is still on the order of $O(n \cdot e)$. In addition, during the computation, two matrices are used and thus more space than Warren's is needed. The time for inserting m edges successively is $O(m \cdot n)$.

Table 2 shows the computational complexities of all the main matrix-based algorithms.

Tabel 2. Comparison of computation complexities

	Warren	*Ibaraki and Katoh*	*Italiano*	*La Poutre and Leeuwen*
time	$O(n \cdot e)$	$O(n \cdot e)$	$O(n \cdot e)$	$O(n \cdot e)$
space	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
path check	$O(1)$	$O(1)$	$O(n)$	$O(1)$
insert m edges	$O(m\alpha(m + e, n) + e + n)$	$O(n^3)$	$O(m \cdot n)$	$O(m \cdot n)$
insert a node	$\ge O(n)$	$\ge O(n)$	$\ge O(n)$	$\ge O(n)$

7. 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), and the method based on graph labeling (the one discussed in this paper). We chose Teuhola's method for testing since it has best space complexity over the others.

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 [43] no formal method was suggested to handle this case using signatures. In fact, even for DAGs, it was not explicitly discussed in [43] how to decompose a digraph into a set of 'spanning' trees (as defined in that article). We code for the specific data setting described in [43]; but it is not a general strategy.

Along with [43], we have tested the following three cases:

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

(2) a forest of 18 trees with four children per nonleaf; eight levels, 393192 nodes, and 393174 connections;

(3) 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. 11. 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 or 64 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 or 22 bits long. For the tests, the buffer is of 50 pages and each page is of size 4096 bytes.

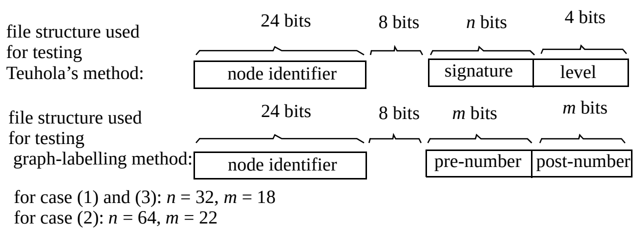

file structure used for testing Teuhola's method:
24 bits
8 bits
n bits
4 bits
node identifier
signature
level
file structure used for testing graph-labelling method:
24 bits
8 bits
m bits
m bits
node identifier
pre-number
post-number
for case (1) and (3): *n* = 32, *m* = 18
for case (2): *n* = 64, *m* = 22

Fig. 11. File structures

Fig. 12 shows the test results of case (1). For this test, each of the signatures for Teuhola's is 32 bits long while each of the preorder (postorder) numbers for the graph-labeling takes 18 bits.

From this, 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, if the signatures become longer, the performance of Teuhola's degrades. See Fig. 13 for illustration. This figure shows the test results of case (2). For this test, each signature is 64 bits long while each preorder (postorder) number takes 22 bits. Especially, there is a sharp increase of page accesses of Teuhola's method from level 6 onwards. It is because in these cases, the accumulated impact of long signatures becomes evident.

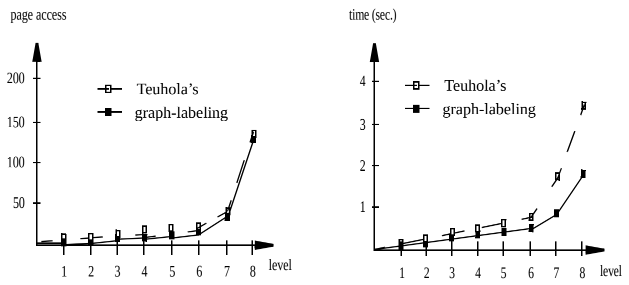


Fig. 12. Test results of case (1)

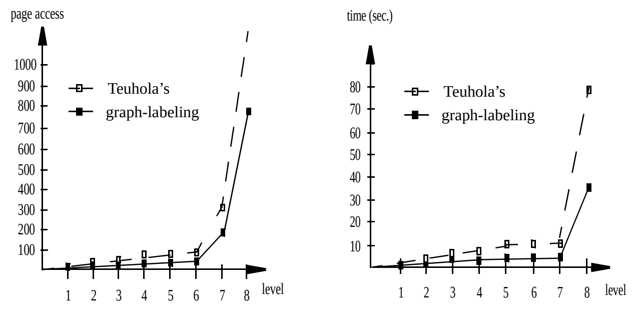


Fig. 13. Test results of case (2)

The test results of case (3) are gathered in Table 1. In this test, all the descendants of a root (3041 nodes on average) are retrieved.

Table 3: Test results of DAGs

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

From Table 3, we can see that in the case of DGAs, Teuhola's method is much worse than ours. It is because a DAG has to be decomposed 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. However, for the graph-labeling method, the graph decomposition is utilized only for the generation of pair sequences, no extra page access is caused. But we notice that in the DAG case, although the number of page access 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.

Fig. 14 shows the test results for the trees with different outdegrees. The forest contains 18 trees each of 3280 nodes. 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 (in 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 [43]. 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 of a graph.

Finally, we emphasize that the academic merit of the proposed method does not only consists in speeding-up the recursion in a relational environment, but also in the new representation of recursive closures, which leads to a better computational complexity than any existing strategies for this problem.

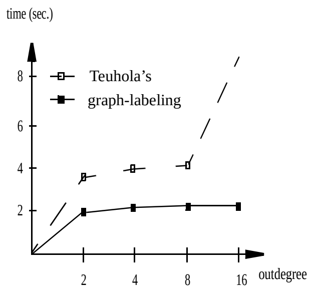


Fig. 14. Test results

8. CONCLUSION

In this paper, a new labeling technique has been proposed. Using this technique, the recursion w.r.t. a tree hierarchy can be computed without join operations. The proposed method can be extended to compute the recursion w.r.t. DAGs and graphs containing cycles. Given a DAG, we can always find a maximal branching as its representation, which is in fact a tree (or a forest) and thus can always be labeled. To find the recursive closure of the DAG, we construct a sequence of label pairs for each node, which can be done in a topological order of the DAG in $O(e \cdot b)$ time and $O(n \cdot b)$ space, where n represents the number of the nodes of the DAG, e the numbers of the edges, and b the DAG's breadth. To compute the recursion w.r.t. a cyclic graph, we first use Tarjan's algorithm [41] to find all its SCCs; and then condense each SCC onto a single node, resulting in a DAG. Since Tarjan's algorithm needs only $O(n + e)$ time, the time complexity for finding the sequences of label pairs for all nodes of a cyclic graph is theoretically bounded by $O(e \cdot b)$. Finally, we notice that the sequences of label pairs are in fact a new representation of the transitive closure of a graph with a much less space overhead than any existing strategy. Therefore, the method discussed in this paper can be considered as a new method for computing transitive closures with optimal time and space complexities.

REFERENCES

[1] 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.

[2] R. Agrawal, A. Borgida and H.V. Jagadish, "Efficient management of transtive relationships in large data and knowledge bases," *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data*, Oregon, 1989, pp. 253-262.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] M.F. van Bommel and T.J. Beck, "Incremental Encoding of Multiple Inheritance Hierarchies Supporting Lattice Operations, *Linkoping Electronic Articles in Computer and Information Science*, http://www.ep.liu.se/ea/cis/2000/001.

[8] 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.

[9] K.S. Booth and G.S. Leuker, "Testing for the consecutive ones property, interval graphs, and graph palanity using PQ-tree algorithms," *J. Comput. Sys. Sci.*, 13(3):335-379, Dec. 1976.

[10] 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.

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

[12] 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.

[13] 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.

[14] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson, "The Modula-3 type system," P*roc. of 16th Simposium on Principles of Programming Languages, POPL'89*, ACM SIGPLAN, Austin, Texas, Jan. 1989, pp. 202-212.

[15] Y. Chen, "On the Graph Traversal and Linear Binary-chain Programs," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 3, May 2003, pp. 573-596.

[16] N.H. Cohen, "Type-extension tests can be performed in constant time," *ACM Transactions on Programming Languages and Systems*, 13:626-629, 1991.

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

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

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

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

[21] J. Ebert, “A Sensitive Transitive closure Algorithm,” *Inf. Process Letters* 12, 5 (1981).

[22] J. Eve and R. Kurki-Suonio, “On Computing the Transitive Closure of a Relation,” *Acta Informatica* 8, 303 - 314, 1977.

[23] 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.

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

[25] G.F. Italiano, Amortized effieciency of a path retrieval data structure, *Theoretical Computer Science*, 48:273-281, 1986.

[26] Y.E. Ioannidis, R. Ramakrishnan and L. Winger, “Transitive Closure Algorithms Based on Depth-First Search,” *ACM Trans. Database Syst.*, Vol. 18. No. 3, 1993, pp. 512 - 576.

[27] H.V. Jagadish, “A Compression Technique to Materialize Transitive Closure,” *ACM Trans. Database Systems*, Vol. 15, No. 4, 1990, pp. 558 - 598.

[28] T. Keller, G. Graefe and D. Maier, “Efficient Assembly of Complex Objects,” *Proc. ACM SIGMOD conf.* Denver, Colo., 1991, pp. 148-157.

[29] W. Kim, “Object-Oriented Database Systems: Promises, Reality, and Future,” *Proc. 19th VLDB conf.*, Dublin, Ireland, 1993, pp. 676-687.

[30] D.E. Knuth, *The Art of Computer Programming, Vol.1*, Addison-Wesley, Reading, 1969.

[31] 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.

[32] A. La Poutre and J. van Leeuwen, Maintenance of Transitive closure and transitive reduction of graphs, in *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 106-120. *Lecture Notes in Computer Science 314*, Springer-Verlag, 1988.

[33] B. Lindsay, J. McPherson and H. Pirahesh, “A Data Management Extension Architecture,” *Proc. ACM SIGMOD conf.*, 1987, pp. 220-226.

[34] K. Mehlhorn, “*Graph Algorithms and NP-Completeness*: *Data Structure and Algorithm 2*” Springer-Verlag, Berlin, 1984.

[35] 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.

[36] P. Purdom, “A Transitive Closure Algorithm,” *BIT* 10, 76 -94, 1970.

[37] R. Ramakrishnan and J.D. Ullman, “A Survey of Research in Deductive Database Systems,” *J. Logic Programming*, May, 1995, pp. 125-149.

[38] L. Schmitz, “An Improved Transitive Closure Algorithm,” *Computing* 30, 359 - 371 (1983).

[39] 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.

[40] M.A. Schubert and J. Taugher, “Determing type, part, colour, and time relationship,” 16 (*special issue on Knowledge Representation*):53-60, Oct. 1983.

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

[42] J. Tarjan: Finding Optimum Branching, *Networks*, 7. 1977, pp. 25 -35.

[43] 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.

[44] S. Warshall, “A Theorem on Boolean Matrices,” *JACM*, 9. 1(Jan. 1962), 11 - 12.

[45] H.S. Warren, “A Modification of Warshall’s Algorithm for the Transitive Closure of Binary Relations,” *Commun. ACM* 18, 4 (April 1975), 218 - 220.

[46] 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.

[47] P. Valduriez, S. Khoshafian and G. Copeland, “Implementation Techniques of Complex Objects,” *Proc. 12th VLDB Conf.*, Kyoto, Japan, 1986, pp. 101-109.

[48] Y. Zibin and J. Gil, “Efficient Subtyping Tests with PQ-Encoding,” *Proc. of the 2001 ACM SIGPLAN conf. on Object-Oriented Programming Systems, Languages and Application*, Florida, October 14-18, 2001, pp. 96-107.

[49] C. Zhang, J. Naughton, D. DeWitt, Q. Luo and G. Lohman, “On Supporting Containment Queries in Relational Database Management Systems, in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, California, USA, 2001.