A New Way To Speed Up Recursion in Relational Databases

Yangjun Chen*
Department of Business Computing, University of Winnipeg, Manitoba, Canada, ychen2@uwinnipeg.ca

ABSTRACT

*Composite object represented as a directed graph is an important data structure which requires efficient sup-port in CAD/CAM, CASE, office systems, software man-agement, web databases and document databases. It is cumbersome to handle such an object in relational database systems when it involves recursion relationships. In this pa-per, we present a new encoding method to support the effi-cient computation of recursion. In addition, we devise a linear time algorithm to identify the sequence of spanning trees (forests) w.r.t. a directed acyclic graph (DAG), which covers all the edges of the graph. Together with the new en-coding method, this algorithm enable us to compute recur-sion w.r.t. a DAG in time O(e), where e represent the number of the edges of the DAG. More importantly, this method is especially suitable for a relational environme*nt.

INTRODUCTION

It is a general opinion that relational database systems are inad-equate for manipulating composite objects which arise in novel appli-cations such as web and document databases [MMM97, ACCM97, CA98, CA99], CAD/CAM, CASE, office systems and software man-agement [BKKG88, Te96]. Especially, when recursive relationships are involved, it is cumbersome to handle them in relational systems.

A composite object can be generally represented as a direct-ed graph. For example, in a CAD database, a composite ob-ject corre-sponds to a complex design, which is composed of several subdesigns [BKKG88]. 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 scien-tific literature, recording reference relationships between au-thors, constructs a directed cyclic graph. As a third example, we consider the traditional organization of a company with a variable number of man-ager-subordinate levels, which can be represented as a tree hierarchy. In a relational system, composite objects must be fragmented across many rela-tions, requiring joins to gather all the parts. A typical ap-proach to improving join efficiency is to equip relations with hid-den pointer fields for coupling the tuples to be joined [Ca90].

Recently, a new method has been proposed by Teu-hola [Te96], in which the information of the ancestor path of each node is packed into a fix-length code, called the signa-ture. Then, the operation to find a transitive closure can be performed by identifying a series of signature intervals. No joins are needed. Using Teuhola's method, CPU time can be improved up to 93% for trees and 45% for DAGs in compar-ison with a method which performs a SELECT command against each node, where the relation to store edges is equipped with a clustering index on the parent nodes [Te96].

In this paper, we follow the method proposed in [Te96], but using a different encoding approach to pack "ancestor paths". For example, in a tree hierarchy, we associate each node *v* with a pair of integers (a, b) such that if *v'*, another node associated with (a*'*, b*'*), is a descendant of *v*, some arithmet-ical relationship between a and a*'*, as well as b and b*'* 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 a DAG or a directed graph containing cycles by decomposing a graph into a sequence of trees (forests), in which the approach described above can be employed. As we can see later, a new method can be developed based on the techniques mentioned above, by which recursion can be evalu-ated in O(*e*) time, just as an algorithm using *adjacency lists*. (The adjacency list is a common data structure to store a graph in computa-tional graph theory [Me84].) However, our method is especially suit-able for the implementation in a relational environment.

TASK DEFINITION

We consider composite objects represented by a directed graph, where nodes stand for objects and edges for parent- child relationships, stored in a binary relation. In many appli-cations, the transitive clo-sure of a graph needs to be comput-ed, which is defined to be all ancestor- descendant pairs. A lot of researches have been directed to this issue. Among them, the semi-naive [BR86] and the logarithmic [VB86] are typi- cal *algorithmic* solutions.

Another main approach is the ma-terialization of the closure, either partially or completely [AJ89, Ja90]. Recently, the implemen-tation of the transitive closure algorithms in a relational environment has received extensive attention, including performance and the adapta-tion of the traditional algorithms [ADJ90, AJ90, IRW93, DR94].

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 special-ized data structure. The following is a typical structure to ac-commodate part-subpart rela-tionship [CS92]:
• Part(Part-id, Part-rest)
• Connection(Parent-id, Child-id, Conn-rest)
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 the ancestor-descendant relationship.

In the rest of the paper, the following three types of graphs 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 re-striction 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.

LABELING A TREE STRUCTURE

In the method proposed in [Te96], 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* £ *l'* and *h* ³ *h'* hold. Although this meth-od is incomparably superiors to a trivial method, it suffers from the following disadvantages:
(1) This method is space-consuming since signatures tend to be very long.
(2) The size of signatures has to be pre-determined. Different applica-tions may require different sig-nature lengths. This can be tuned only manually.

(3) There may exist the so-called signature conflicts, i.e., two nodes may be assigned the same signature.

In the following, we search for remedies for these three draw-backs. First, we discuss a tree labeling method to demon-strate the main idea of the improvement in this section. The discussion on general cases will be postponed to Section 4.

Consider a tree *T*. 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 the same way, by traversing *T* in *postorder*, each node will get another number *post*(*v*). These two numbers can be used to characterize the ancestor-descendant relationship 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 [Kn73].

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 de-scendant 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. 1(a). The first element of each pair is the preorder number of the corresponding node and the second is the postorder number of it. Using such labels, the ancestor-descendant relationship can be easily checked.

Figure 1: Labeling a tree

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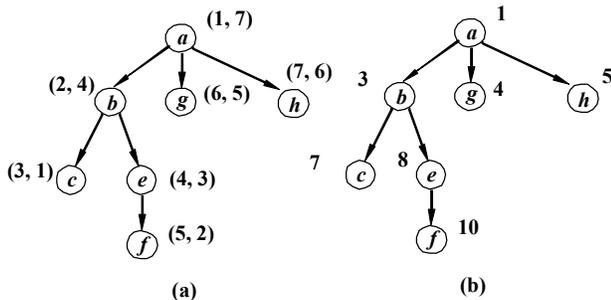
For example, by checking the label associated with *b* against the label for *f*, we know that *b* is an ancestor of *f* in terms of Theorem 1. We can also see that since the pairs associated with *g* and *c* do not satisfy the condition given in Theorem 1, *g* must not be an ancestor of *c* and *vice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

According to this labeling strategy, the relational schema to handle recursion can consists of only one relation of the fol-lowing form:

Node(Node_id, label_pair, Node_rest),

where label_pair is used to accommodate the preorder num-ber and the postorder number of each node, denoted label_pair.preorder and label_pair.postorder, respectively. Then, to retrieve the descendants of node *x*, we issue two queries. The first query is very simple as shown below:

SELECT   label_pair
FROM   Node
WHERE   Node_id = *x*

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

SELECT   *
FROM   Node
WHERE   label_pair.preorder > *y*.preorder
             and label_pair.postorder < *y*.pos-torder

From the above discussion, we can see that the three draw-backs of Teuhola's method [Te96] mentioned above can be eliminated: (1) each node is associated with only a pair of integers and therefore the

space overhead is low; (2) the size of each label pair remains the same for all applications; (3) there is no signature conflicts since each label pair is differ-ent from the others.

In the following, we show another two important technique to identify the sibling relationship and the parent-child rela-tionship.

For the first task, consider a new labeling method as shown in Fig. 1(b). First we assign 1 to the root; then during the breadth-first tra-versal we number the children of each node consecutively from *x* + 2, where *x* is the largest number as-signed so far. We call such a labeling method the sibling-code. Then, we can associate each parent node with an inter-val [*a*, *b*] such that each child's sibling-code *s* ∈ [*a*, *b*]. There-fore, two nodes are siblings iff their sibling-codes be-long to the same interval.

To identify the parent-child relation, we associate each node with a level number. The root has the level number 0. All the children of the root have the level number 1, and so on. Then, if node *x* is the ancestor of *y* and at the same time *l*(*x*) = *l*(*y*) - 1 (*l*(*x*) stands for the level number of *x*), then we know that *x* is the parent of *y*.

GENERALIZATION

Now we discuss how to treat the recursion w.r.t. a general struc-ture: a DAG or a graph containing cycles. First, we ad-dress the prob-lem with DAGs in 4.1. Then, the cyclic graphs will be discussed in 4.2.

Recursion w.r.t. DAGs

We want to apply the technique discussed above to a DAG. To this end, we divide a DAG into a set of spanning trees (forests). This method shares the flavor of Teuhola's [Te96].

But our decomposition strategy is quite different from [Te96]. In [Te96], a DAG is decomposed into a set of spanning trees which are separated from each other, i.e., there are no com-mon nodes between any two spanning trees while in ours two spanning trees (forests) may have common nodes. The advan-tage of our method can be seen in the following discussion.

In the following, we concentrate on only on single-root graphs for simplicity. But the proposed method can be easily extended to normal cases. We construct a sequence of span-ning trees (forests) w.r.t. a DAG *G* with the single-root *r*, which covers all the edges of *G*. First, we extract the spanning tree (forest) from *G,* which contains all nodes of *G*, denoted $T_{max}(G)$. Then, we remove $T_{max}(G)$ and subsequently all iso-lated nodes from *G*, getting another graph G_1. Next, we con-struct a spanning tree (forest) 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 $O(k(n + e))$ time by repeating graph search procedure *k* times, where *n* and *e* represent the number of the nodes and the edg-es of the DAG, respectively. However, this time complexity can be reduced to $O(n + e)$ by implementing an algorithm which com-putes such a sequence in a single-scan.

For a DAG *G* = (*V*, *E*), we represent the sequence of spanning trees (forests) $T_{max}(G_i)$ (*i* = 0, 1, ..., *m*; $G_0 = G$) as follows:

$T_{max}(G_0) = (V_1, E_1)$,
$T_{max}(G_1) = (V_2, E_2)$,
$T_{max}(G_2) = (V_3, E_3)$,
... ...
$T_{max}(G_m) = (V_{m+1}, E_{m+1})$,

where V_1 stands for the set of nodes in *G*, V_i (*i* = 2, ..., *m*+1) for the set of nodes in $G - E_1 \cup E_2 \cup ... \cup E_{i-1}$, and *m* is the largest in-degree of the nodes of *G*.

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

The idea is to construct all $E_1, E_2, ... E_m$ in a single scan. Dur-ing the graph search we compute, for each edge *e* being scanned, the *i* satisfying *e* 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 algo-rithm, we always chose an unvisited edge *e* that is adjacent to edge *e'* ∈ E_i with the largest *i*. In the algorithm, we associate each

node v with a label $l(v)$: $l(v) = i$ indicates that v has been reached by an edge of the forest $T_{max}(G_{i-1}) = (V_i, E_i)$. 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
    E₁ := E₂ := ... := Eₘ := ∅;
    Mark all nodes v ∈ V and all edges e∈ E "unvisited";
    l(v) := 0 for all v ∈ V;
    while there exist "unvisited" nodes do
      begin
        choose an "unvisited" node v ∈ 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 (≠ v);
*           E_{l(u)+1} := E_{l(u)+1} ∪ {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 the edges of three span-ning trees shown in Fig. 2(b). In Appendix, we will trace the execution of the algorithm against Fig. 2(a) for a better un-derstanding.

In the above algorithm, each edge is visited exactly once. There-fore, the time complexity of the algorithm is bounded by $O(n + e)$. In the following, we prove a theorem to estab-lish the correctness of the algorithm.

Figure 2: DAG and its node-disjunct maxiaml

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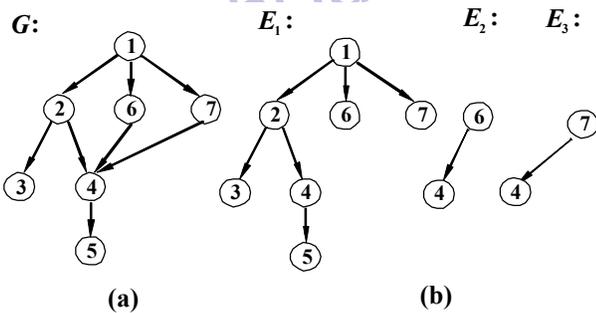
Proposition 2. Applying Algorithm *"find-forest"* to a DAG G, a sequence of spanning trees (forests) w.r.t. G will be found, which covers all of its edges.

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 theorem, we need now to specify that in every E_i, except the root nodes 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 differ-ent 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 theorem proof.

Now we can label each E_i in the same way as discussed in the previous section. (A forest can be regarded as a tree with a virtual root which has a virtual edge linking each tree root of the forest.) In addition, we notice that a node may appear in several E_i's. For ex-ample, in Fig. 2(b) node 6 appears in E_1 and E_2 while node 4 occurs in all the three spanning trees. Then, after labeling each E_i, each node v will get a pair se-quence 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 in-degree of v.,( ) stands for the preorder number and pos-torder number of v w.r.t E_i. In the subsequent discussion, we also say that a label belongs to some E_i, referring to the fact that this pair is calculated in terms of E_i. In terms of such a data structure, we give a naive algorithm below.

$\Delta_{global} := \emptyset;$

$\Delta_{local} := \emptyset;$

$S := \{x\};$ (* The descendants of x will be searched. *)

function *recursion*(S)

begin

  for each $x \in S$ **do** {

    let $p_1.p_2. ... p_m$ be the pair sequence associated with x;

    for $i = m$ to 1 **do** {

*   let Δ be the set of descendants of x w.r.t. E_i

      (evaluated using p_i);

**   for each $y \in \Delta$, remove the pair belonging to E_i from the pair sequence associated with y;

      $\Delta_{local} := \Delta_{local} \cup \Delta;$}}

  $\Delta_{local} := \Delta_{local} - \Delta_{global};$

  $\Delta_{global} := \Delta_{global} \cup \Delta_{local};$

  call *recursion*$(\Delta_{local});$

end

In the above algorithm, pay attention to the line marked with *, by which all the descendants of x will be evaluated in E_i using p_i. Since these descendants may appear also in other E_j's, they should be used for the further computation. But the pair belonging to E_i has to be eliminated from the pair se-quences associated with these nodes to avoid the repeated ac-cess to the edges in E_i, which is done by the line marked with **.

The above algorithm suffers, however, from redundancy as dis-cussed below.

The graph shown in Fig. 3(a) can be decomposed into two span-ning trees as shown in Fig. 3(b). Applying *recursion*(7) to this graph, the descendant set evaluated in the first **for** loop is $\{4, 5\}$. In the second **for** loop, the descendants of node 4 and 5 will be computed, which are $s_1 = \{5, 6\}$ (the descen-dants of node 4) and $s_2 = \{6\}$ (the

Figure 3: Illustration of redundancy of recursion(S)

(a)   **(b)**
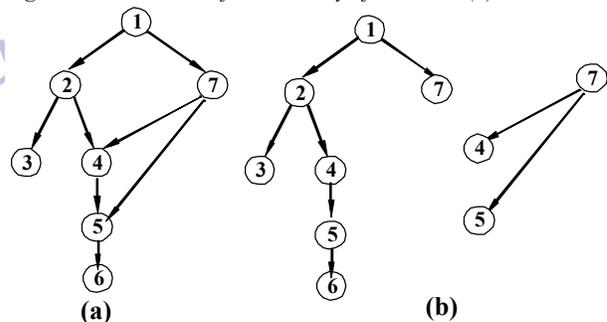
descendants of node 5), re-spectively. Obviously, s_2 is completely covered by s_1. Therefore, the work of evaluating s_2 can be saved. To this end, we associate each E_i with a bit string of size n, denoted B_i. If some node j is a descendant evaluated w.r.t. E_i, the jth bit of B_i will be set to 1, i.e., $B_i[j] = 1$. If the descendants of a node k w.r.t. E_i will be evaluated, we first check $B_i[k]$ to see wheth-er it is equal to 1. If so, the corresponding computation will not be made. Another problem is that if s_2 is evaluated first, the redundant work can not be avoided even though the checking is performed. Thus, the order of the nodes whose descendants will be evaluated is important. For an E_i the nodes with smaller preorder numbers will be treated earlier than those with a larger preorder number. It is because a node with a larger preorder number may be a descendant of a node with a smaller one, but not *vice versa*. In order to sort the nodes in this way, we have to change the control method of the above algorithm. Assume that each node v is associated with a pair sequence of the form: $p_1 \cdot p_2 \cdot \ldots \cdot p_m$, where m is the largest in-degree of the graph. If v does not appear in E_i, p_i will be of the form: (_, _) and will be ignored by sorting. The nodes, whose descendants w.r.t. E_i are going to be evaluated, will be first sorted in terms of p_i. Then, the descendants of these nodes w.r.t. E_i will be computed. In a second loop, the nodes will be sorted again in terms of p_{i-1}. This process re-peats until all p_i's are handled. Below is the corresponding al-gorithm with the checking mechanism used.

$\Delta_{global} := \varnothing$;
$\Delta_{local} := \varnothing$;
$S := \{x\}$;   (* The descendants of x will be searched. *)
let $p_1 \cdot p_2 \cdot \ldots \cdot p_m$ be the pair sequence associated with each node of the graph;
for $i = 1$ to m **do** $B_i = 0$;
function *refined-recursion*(S)
begin
   for $i = m$ to 1 **do** {
      sort S in terms of p_i's;
      let the sorted S be $\{v_1, \ldots, v_k\}$;

      for $j = 1$ to k **do** {

         if $B_i[v_j] = 0$ then $\Delta :=$ the set of descendants of v_j

                        w.r.t. E_i (evaluated using p_i);
         for each $v_j \in \Delta$ **do** $\{B_i[v_j] := 1\}$
         $\Delta_{local} := \Delta_{local} \cup \Delta;\}\}$
      $\Delta_{local} := \Delta_{local} - \Delta_{global}$;
      $\Delta_{global} := \Delta_{global} \cup \Delta_{local}$;
      call *refined-recursion*(Δ_{local});

e n d

Note that we take only O(1) time to check a bit in the bit string. For each newly evaluated node set (each time stored in Δ_{local} in the above algorithm), sorting operations will be per-formed. But each node v in Δ_{local} can take part in the sorting only d times, where d represents the in-degree of v, since for each node v only d pairs in the pair sequence associated with it is not of the form: (_, _). Assume that each time only Δ_{ij} from $\Delta_i (= \Delta_{local})$ participates in the sorting. Then, the total cost for sorting is

$$\sum_i \sum_j | \Delta_{ij} | \cdot \log | \Delta_{ij} | \leq e \cdot \log n$$

Since each edge is visited at most once, the traversal of the graph needs only O(e) time. Therefore, the time complexity of the algo-rithm is bounded by O($e \times \log n$), a little bit more than the time required by an algorithm using an adjacency list. But this algorithm is quite suitable for a relational environment. Furthermore, we can store the data in a special way to support the sorting operation so that no extra time is required. For ex-ample, we can define two simple relations to accommodate the graph and its pair sequences as follows:
   node(Node_id, Node_rest),
   spanning_forest(E_num, label_pair, Node_id).

The first relation stores all the nodes of the graph. The second relation stores all the spanning trees (forests), in which "E_num" is for the identifiers of the spanning trees (forests). If for each of them the label pairs are stored in the increasing order of their preorder numbers, the sorting operations in the algorithm *refined-recursion*() can be removed. Then, the time complexity of the algorithm can be reduced to O(e). This can be done as follows. Whenever some E_i is considered during the execution, we take the tuples with E_num = i from the re-lation "spanning_forest". Then, we scan these tuples and check, for each tuple, to see whether B_i[node_id] = 1. If it is the case, the corresponding label pair will be put in a list (a temporary data structure) sequentially. Obviously, the list constructed in this way is sorted into the increasing order of the preorder numbers w.r.t. E_i.

Recursion w.r.t. Cyclic Graphs

Based on the method discussed in the previous section, we can easily develop an algorithm to compute recursion for cy-clic graphs. We can use Tarjan's algorithm for identifying strong connected com-ponents (SCC) to find cycles of a cyclic graph [Ta72] (which needs only O($n + e$) time). Then, we think of each SCC as a single node (i.e., condense each SCC to a node). The resulting graph is a DAG. Applying the algo-rithm *find_forest*( ) to this DAG, we will get a set of forests. For each forest, we can associate each node with a pair as above. Obviously, all nodes in an SCC will be assigned the same pair (or the same pair sequence). For this reason, the method for evaluating the recursion at some node x should be changed. For example, if a graph becomes a tree after con-densing each SCC to a node, the select-from-where state-ments like those given in Section 3 (against this graph) can be modified as follows. The first query is quite the same as that shown in Section 3:

   SELECT   label_pair
   FROM   Node
   WHERE   Node_id = x
But the second is changed slightly:
   SELECT   *
   FROM   Node
   WHERE   label_pair.preorder ³ y.preorder
                   and label_pair.postorder £ y.pos-torder
By the second query, the nodes in the same SCC as x will be regarded as the descendants of x.

For general cases, the method for checking ancestor-descen-dant relationship applies. No extra complexity is caused. Since Tarjan's algorithm runs in O($n + e$) time, computing re-cursion for a cyclic graph needs only O(e) time.

CONCLUSION

In this paper, a new labeling technique has been proposed. Using this technique, the recursion w.r.t. a tree hierarchy can be evaluated very efficiently. In addition, we have introduced a new algorithm for computing spanning trees (forests), which requires only linear time. Together with the labeling technique, this method enable us to develop an efficient algo-rithm to compute recursion for directed graphs in O(e) time, where e represent the number of the edges of the DAG. More importantly, this method is especially suitable for relational databases and much better than the existing methods.

REFERENCES

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

ADJ90 R. Agrawal, S. Dar, H.V. Jagadish, "Direct transi-tive closure algorithms: Design and performance evaluation," *ACM Trans. Data-base 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 Engi-neering*, Los Ange-les, 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, "Clus-tering 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 Strat-egies," in: *Proc. ACM SIGMOD Conf.*, Washing-ton D.C., 1986, pp. 16 - 52.

Ca95 M. Carey et al., "An Incremental Join Attachment for Starburst," in: *Proc. 16th VLDB Conf.,* Bris-bane, Australia, 1990, pp. 662 - 673.

CA98 Y. Chen, K. Aberer, "Layered Index Structures in Document Database Systems," *Proc. 7th Int. Con-ference on Information and Knowledge Manage-ment (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*, Flo-rence, Italy: Springer Verlag, Sept. 1999. pp. 473 - 484.

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

DR94 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.

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 Ma-terilize Transitive Closure," ACM Trans. Database Systems, Vol. 15, No. 4, 1990, pp. 558 - 598.

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

Me84 K. Mehlhorn, "*Graph Algorithms and NP-Com-pleteness*: *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.

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.

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

APPENDIX

In this Appendix, we trace the algorithm *find-spanning-tree* against the tree shown in Fig. 2(a).

See Fig. 4. At the beginning, every $r(v)$ is set to 0. After the first loop, the l-value of node 1 remains 0. But the l-values of 2, 6, and 7 are changed to 1. Moreover, node 1, and edge (1, 2), (1, 6) and (1, 7) 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. 5, 6, 7 and 8.

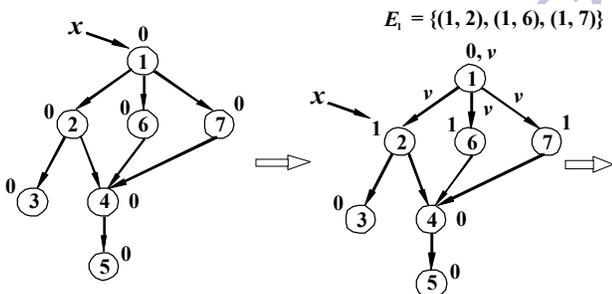*Figure 4: The first execution step of find-node-disjunct-forest*

$E_1 = \{(1, 2), (1, 6), (1, 7)\}$

Figure 5: The second and third execution step of find-node-disjunct-forest

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4)\}$

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4)\}$
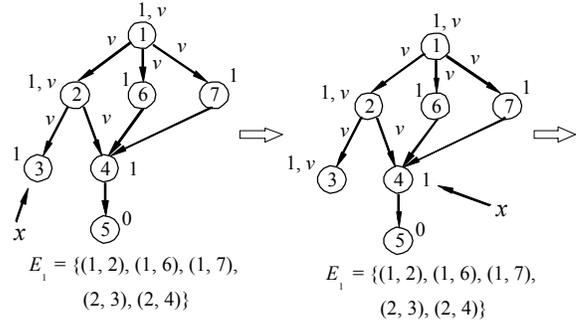
Figure 6: The fourth and fifth execution step of find-node-disjunct-forest

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4), (4, 5)\}$

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4), (4, 5)\}$
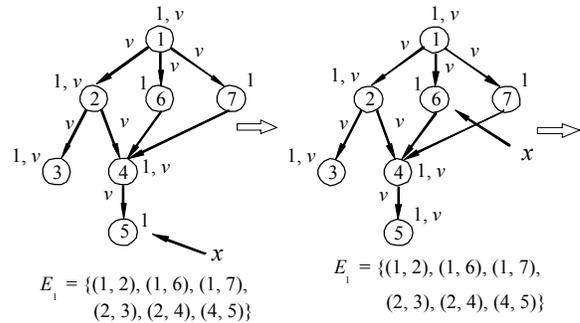
Figure 7: The fourth and fifth execution step of find-node-disjunct-forest

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4), (4, 5)\}$

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4), (4, 5)\}$
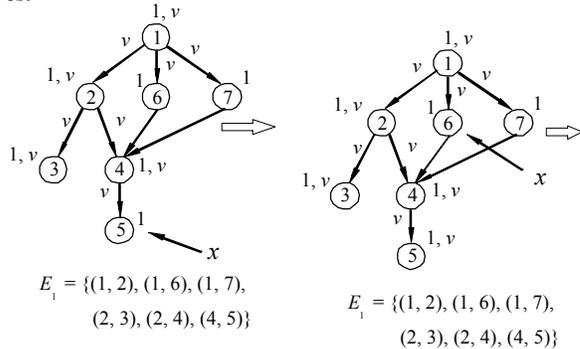
Figure 8: The sixth and seventh execution step of find-node-disjunct-forest
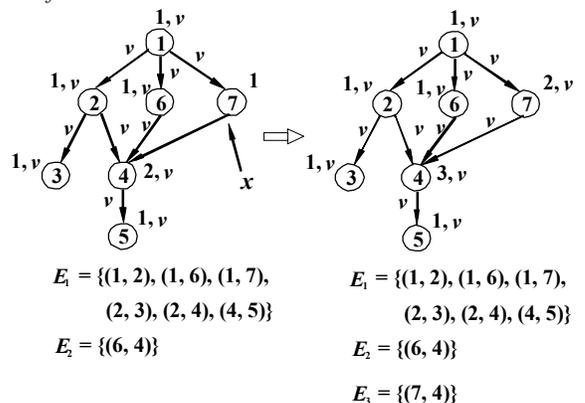
$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4), (4, 5)\}$

$E_2 = \{(6, 4)\}$

$E_1 = \{(1, 2), (1, 6), (1, 7),$
$(2, 3), (2, 4), (4, 5)\}$

$E_2 = \{(6, 4)\}$

$E_3 = \{(7, 4)\}$

Related Content

Tradeoffs Between Forensics and Anti-Forensics of Digital Images
Priya Makarand Shelke and Rajesh Shardanand Prasad (2017). *International Journal of Rough Sets and Data Analysis (pp. 92-105).*
www.irma-international.org/article/tradeoffs-between-forensics-and-anti-forensics-of-digital-images/178165

An Efficient Intra-Server and Inter-Server Load Balancing Algorithm for Internet Distributed Systems
Sanjaya Kumar Panda, Swati Mishra and Satyabrata Das (2017). *International Journal of Rough Sets and Data Analysis (pp. 1-18).*
www.irma-international.org/article/an-efficient-intra-server-and-inter-server-load-balancing-algorithm-for-internet-distributed-systems/169171

Stock Price Trend Prediction and Recommendation using Cognitive Process
Vipul Bag and U. V. Kulkarni (2017). *International Journal of Rough Sets and Data Analysis (pp. 36-48).*
www.irma-international.org/article/stock-price-trend-prediction-and-recommendation-using-cognitive-process/178161

Thinking Outside the Office: The Impact of Virtual Work on Creative Workers' Attitudes
Beth A. Rubin and April J. Spivack (2012). *Virtual Work and Human Interaction Research (pp. 59-77).*
www.irma-international.org/chapter/thinking-outside-office/65315

Covering Based Pessimistic Multigranular Approximate Rough Equalities and Their Properties
Balakrushna Tripathy and Radha Raman Mohanty (2018). *International Journal of Rough Sets and Data Analysis (pp. 58-78).*
www.irma-international.org/article/covering-based-pessimistic-multigranular-approximate-rough-equalities-and-their-properties/190891