Graph Encoding and Recursion Computation

Yangjun Chen
University of Winnipeg, Canada

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 (Abiteboul, Cluet, Christophides, Milo, Moerkotte & Simon, 1997; Chen & Aberer, 1998, 1999; Mendelzon, Mihaila & Milo, 1997; Zhang, Naughton, Dewitt, Luo & Lohman, 2001), CAD/ CAM, CASE, office systems and software management. 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 (Kuno & Rundensteiner, 1998; Lee & Lee, 1998). To overcome this problem, a lot of interesting graph encoding methods have been developed to mitigate the difficulty to some extent. In this article, we give a brief description of some important methods, including analysis and comparison of their space and time complexities.

BACKGROUND

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. Often, subdesigns are shared by more than one higher-level design, 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. The so-called *join index* is another auxiliary access path to mitigate this difficulty. Also, several advanced join algorithms have been suggested, based on hashing and a large main memory. In addition, a different kind of attempt 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. Another extension to relational system is *nested relations* (or NF^2 relations). 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 databases* and *object-relational databases* can be considered as two quite different extensions to handle this problem (Chen, 2003; Ramakrishnan & Ullman, 1995).

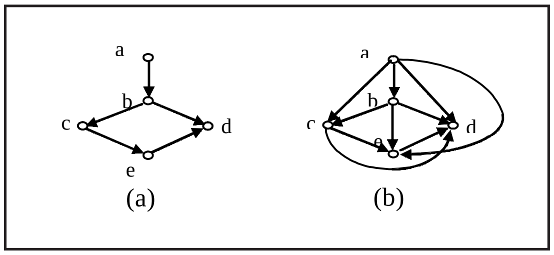
In the past decade, a quite different kind of research has also been done to avoid *join* operations based on *graph encoding*. In this article, we provide an overview on most important techniques in this area and discuss a new encoding approach to pack "ancestor paths" in a relational environment. It 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 (Schmitz, 1983), the graph encoding (Abdeddaim, 1997; Bommel & Beck; Zibin & Gil, 2001) and the matrix-based algorithms (La Poutre & Leeuwen, 1988).

RECURSION COMPUTATION IN RELATIONAL DATABASES

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

In this article, we mainly overview the graph encoding in a relational environment. The following is a typical structure to accommodate part-subpart relationships (Cattell & Skeen, 1992):

Figure 1. A graph and its transitive closure



- Part(Part-id, ...)
- Connection(Parent-id, Child-id, ...)

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

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

(i) Tree hierarchy, in which the parent-child relationship is of one-to-many type; that is, 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 will use the term *graph* to refer to the *directed graph,* since we do not discuss non-directed ones at all.

RECURSION WITH RESPECT TO TREES

Perhaps the most elegant algorithm for encoding is *relative numbering* (Schmitz, 1983), which guarantees both optimal encoding length of log$n$ bits and constant time recursion tests for trees. Consider a tree T. By traversing T in *postorder,* each node v will obtain a number (it can be integer or a real number) $post(v)$ to record the order in which the nodes of the tree are visited. A basic property of postorder traversal is

$$post(v) = \max\{ post(u) \mid u \in \text{descendant}(v)\}.$$

Let $l(v)$ be defined by

$$l(v) = \min\{ post(u) \mid u \in \text{descendant}(v)\}.$$

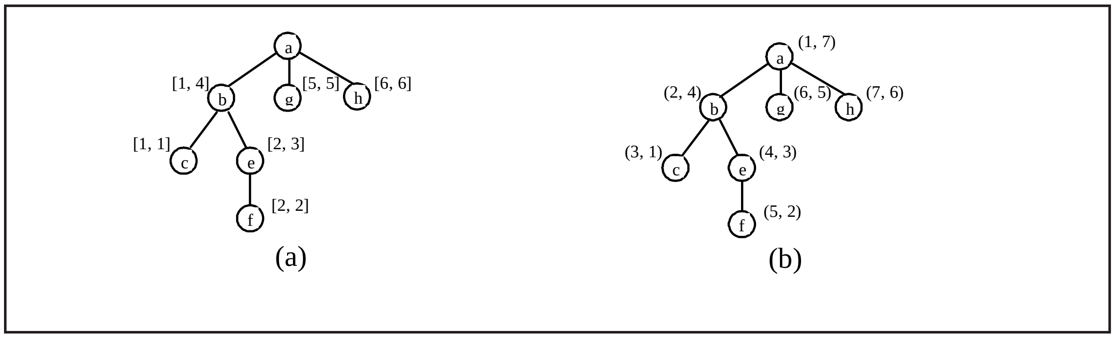
Then, a node u is a descendant of v if $l(v) \leq post(u) \leq post(v)$.

In terms of this relationship, each node v can be encoded by an interval $[l(v), post(v)]$ as exemplified by Figure 2(a).

Another interesting graph encoding method is discussed in Knuth (2003), by which each node is associated with a pair $(pre(v), post(v))$, where $pre(v)$ represents the preorder number of v and can be obtained by traversing T in a preorder. The pair 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 if $pre(v') > pre(v)$ and $post(v') < post(v)$.
Proof. See Exercise 2.3.2-20 in Knuth (1969).

Figure 2. Labeling a tree



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 $post(v)$.

Figure 2(b) helps for illustration. 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.

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').

According to the tree labeling discussed previously, the relational schema to handle recursion can consist 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. If a tree is stored in a preorder, we can control the search so that it stops when the first pair that is not subsumed by y is encountered.

RECURSION WITH RESPECT TO DGAS

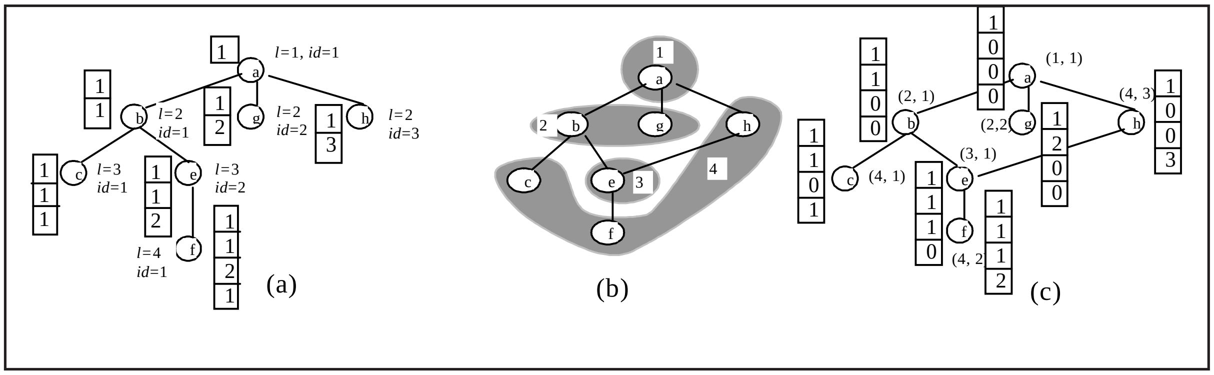
In this section, we consider the recursion computation w.r.t. DAGs.

- *Range-compression*
  The relative numbering was extended to handle DAGs by (Agrawal, Borgida & Jagadish, 1989). Their method is called *range-compression* encoding. It encodes each node v in a DAG as an integer $id(v)$, obtained by scanning a certain spanning forest of the DAG in postorder. In addition, each node v is associated with an array A of length k ($1 \le k \le n$), in which the ith entry is an interval $[l_i, r_i]$ with the property that a node u is a descendant of v if there exists an integer j such that $l_i \le id(u) \le r_j$. Since an array contains n entries in the worst case, this encoding method needs $O(n^2)$ space and time.
- *Cohen's encoding and EP encoding*
  In the language research, the graph encoding is also extensively explored for type-subtype checking that is in essence a recursion computation. Thus, the methods proposed in that area can be employed in a relational environment. Perhaps the most interesting method is Cohen's encoding for tree structures (Cohen, 1991). It was generalized to DAGs by Krall, Vitek and Horspool (1997) into what is called *packed encoding* (PE).

Cohen' encoding stores with each node v its level l_v, its unique identifier id_v, as well as an array A_v such that for each node $u \in$ ancestor(v), $A_v[l_u] = id_u$. The test $v \in$ descendant(u) is then carried out by checking whether both $l_v \ge l_u$ and $A_v[l_u] = id_u$ hold. An example of the actual encoding is given in Figure 3(a).

EP encoding partitions a DAG into a number of slices: $S_1, ..., S_k$ for some k such that no two ancestors of a node can be on the same slice. In addition, each node v is assigned a unique identifier id_v within its slice S. Thus, v is identified by the pair (s_v, id_v), where s_v is the number for S. Furthermore, each node v is associated with an array A_v such that for all $u \in$ ancestor(v), $A_v[s_u] = id_v$. The DAG shown in Figure 3(b) is partitioned into four slices numbered 1, 2, 3 and 4, respectively. Accordingly, the DAG can be encoded as shown in Figure 3(c). We note that in EP encoding slices play a role similar to that of levels in Cohen's encoding. In fact, Cohen's algorithm partitions

Figure 3. Cohen's encoding and EP encoding



a tree into levels while EP encoding partitions a DAG into slices. According to Fall (1995), it is NP-hard to find a minimal partition of slices. Moreover, if the sizes of slices is bounded by a constant, the array associated with a node is of length O(n) at average. So the space and time overhead of EP encoding is on the order of O(n^2).

- *Pre-postorder encoding*
  Now we discuss a new encoding method, which 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.

What we want is to apply the pre-postorder encoding discussed previously to a DAG. To this end, we establish a *branching* of the DAG as follows (Tarjan, 1977).

Definition 2 - (*branching*) 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 Figure 4(b) are a maximal branching of the graph shown in Figure 4(a) if each edge has the 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 Figure 4(c) is the representation of the graph shown in Figure 2(a). Using Tarjan's algorithm for finding optimum branchings (Tarjan, 1977), 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.

Figure 4. A DAG and its branching

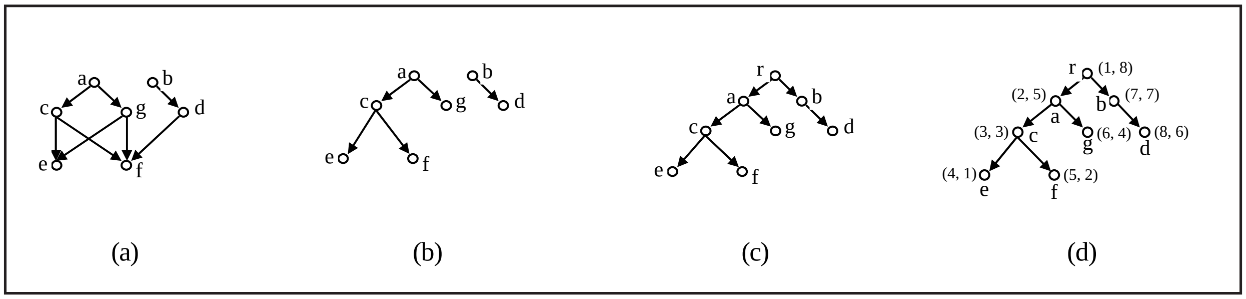
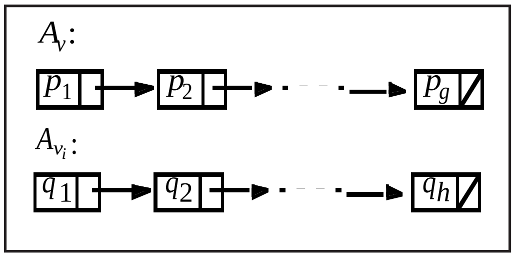


Figure 5. Linked lists associated with nodes in G



We can label G_r in the same way as shown in the previous subsection. See Figure 4(d).

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 show 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.)

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. For this purpose, we sort the nodes of G topologically; that is, $(v_i, v_j) \in E$ 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 for the child node v_l (l = 1, ..., k) as follows. Assume $A_v = p_1 \rightarrow p_2 \rightarrow \ldots p_g$ and A_{v_i} = $q_1 \rightarrow q_2 \rightarrow \ldots\ q_h$, as shown in Figure 5. Assume that both A_v and A_{v_i} are increasingly ordered. (We say a pair p is larger than another pair p', denoted $p > p$' if $p.pre > p'.pre$ and $p.post > p'.post$.)

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

Algorithm *pair-sequence-merge*(A_i, A_{v_i} **);**

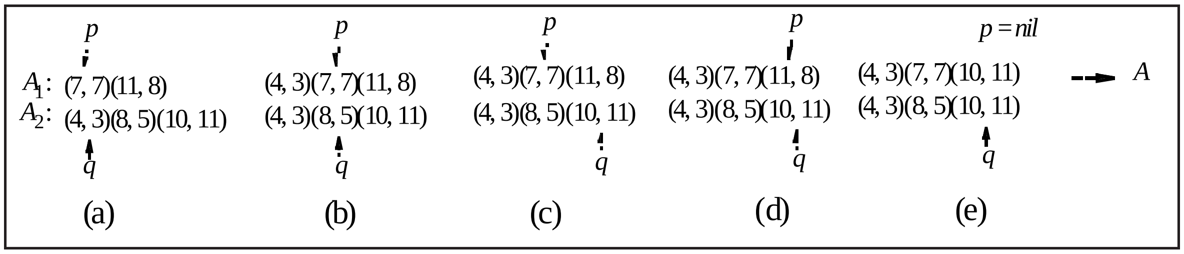
(1) If $p_i.pre > q_j.pre$ and $p_i.post > q_j.post$, insert q_j into A_v after p_{i-1} and before p_i and move to q_{j+1}.
(2) If $p_i.pre > q_j.pre$ and $p_i.post < q_j.post$, remove p_i from A_v and move to p_{i+1}. (*p_i is subsumed by q_j.*)
(3) If $p_i.pre < q_j.pre$ and $p_i.post > q_j.post$, 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_i} .*)
(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.
(6) If $p_i = nil$ and $q_j \neq nil$, attach the rest of A_{v_i} to the end of A_v.

The following example helps for illustration.

Example 1 - Assume that A_1 = (7, 7)(11, 8) and A_2 = (4, 3)(8, 5)(10, 11). Then, the result of merging A_1 and A_2 is (4, 3)(7, 7)(10, 11). Figure 6 shows the entire merging process.

In each step, the A_1-pair pointed to by p and the A_2-pair pointed to by q are compared. In the first step, (7, 7) in A_1 will be checked against (4, 3) in A_2 (see Figure 6(a)). Since (4, 3) is smaller than (7, 7), it will be inserted into A_1 before (7, 7) (see Figure 6(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 Figure 6(c)). In the third

Figure 6. An entire merging process



step, (7, 7) is smaller than (10, 11) and we move to (11, 8) in A_1 (see Figure 6(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 Figure 6(e)). In this case, (10, 11) will be attached to A_1, forming the result $A = (4, 3)(7, 7)(10, 11)$ (see Figure 6(e)).

We can store physically the label pair for each node, as well as its label pair. 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*.

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 (Tarjan, 1972) (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 earlier. 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.

FUTURE TRENDS

The computation of transitive closures and recursive relationships is a classic problem in the graph theory and has a variety of applications in data engineering, such as CAD/CAM, office systems, databases, programming languages and so on. For all these applications, the problems can be represented as a directed graph with the edges being not labelled, and can be solved using the techniques described in this article. In practice, however, there exists another kind of problem, which can be represented only by using the so-called weighted directed graphs. For them, the edges are associated with labels or distances and the shortest (or longest) paths between two given nodes are often asked. Obviously, these techniques are not able to solve such problems. They have to be extended to encode path information in the data structure to speed up query evaluation. For this, an interesting issue is how to maintain minimum information but get high efficiency, which is more challenging than transitive closures and provides an important research topic in the near future.

CONCLUSION

In this article, we provide an overview on the recursion computation in a relational environment and present a new encoding method to label a digraph, which is compared with a variety of traditional strategies as well as the methods proposed in the database community. Our 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 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.

REFERENCES

Abdeddaim, S. (1997). On incremental computation of transitive closure and greedy alignment. In A. Apostolico & J. Hein (Eds.), *Proceedings of 8th Symp. Combinatorial Pattern Matching* (pp. 167-179).

Abiteboul, S., Cluet, S., Christophides, V., Milo, T., Moerkotte, G., & Simon, J. (1997, April). Querying documents in object databases. *International Journal of Digital Libraries, 1*(1), 5-19.

Agrawal, R., Borgida, A., & Jagadish, J.V. (1989, June). Efficient management of transitive relationships in large data and knowledge bases. *Proceedings of the ACM SIGMOD Intl. Conf. on the Management of Data* (pp. 253-262).

Booth, K.S., & Leuker, G.S. (1976, December). Testing for the consecutive ones property, interval graphs, and graph palanity using PQ-tree algorithms. *Journal of Computer Sys. Sci., 13*(3), 335-379.

Chen, Y. (2003, May). On the graph traversal and linear binary-chain programs. *IEEE Transactions on Knowledge and Data Engineering, 15*(3), 573-596.

Chen, Y., & Aberer, K. (1998). Layered index structures in document database systems. *Proceedings of 7th Int. Conference on Information and Knowledge Management (CIKM)*, Bethesda, MD (pp. 406-413). ACM.

Chen, Y., & Aberer, K. (1999, September). Combining pat-trees and signature files for query evaluation in document databases. *Proceedings of 10th Int. DEXA Conf. on Database and Expert Systems Application*, Florence, Italy (pp. 473–484). Springer Verlag.

Cohen, N.H. (1991). Type-extension tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems, 13,* 626-629.

Cattell, R.G.G., & Skeen, J. (1992). Object operations benchmark. *ACM Trans. Database Systems, 17*(1), 1 -31.

Fall, A. (1995). Sparse term encoding for dynamical taxonomies. *Proceedings of 4th International Conf. On Conceptual Structures (ICCS-96): Knowledge Representation as Interlingua,* Berlin (pp. 277-292).

Knuth, D.E. (1969). *The art of computer programming* (vol. 1). Reading, MA: Addison-Wesley.

Krall, A., Vitek, J., & Horspool, R.N. (1997). Near optimal hierarchical encoding of types. In M. Aksit & S. Matsuoka (Eds.), *Proceedings of 11th European Conf. on Object-Oriented Programming*, Jyvaskyla, Finland (pp. 128-145).

Kuno, H.A., & Rundensteiner, E.A. (1998). Incremental maintenance of materialized object-oriented views in MultiView: Strategies and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering, 10*(5), 768-792.

La Poutre, J.A., & van Leeuwen, J. (1988). Maintenance of transitive closure and transitive reduction of graphs. *Proceedings of Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, 314,* 106-120. Springer-Verlag.

Lee, W.C., & Lee, D.L. (1998). Path dictionary: A new access method for query processing in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering, 10*(3), 371-388.

Mendelzon, A.O, Mihaila, G.A., & Milo, T. (1997, April). Querying the World Wide Web. *International Journal of Digital Libraries, 1*(1), 54-67.

Ramakrishnan, R., & Ullman, J.D. (1995, May). A survey of research in deductive database systems. *Journal of Logic Programming*, 125-149.

Schmitz, L. (1983). An improved transitive closure algorithm. *Computing, 30,* 359 - 371.

Stonebraker, M., Rowe, L., & Hirohama, M. (1990). The implementation of POSTGRES. *IEEE Trans. Knowledge and Data Eng., 2*(1), 125-142.

Tarjan, R. (1972, June). Depth-first search and linear graph algorithms. *SIAM J. Compt., 1*(2), 146-140.

van Bommel, M.F., & Beck, T.J. (2000). 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*

Zhang, C., Naughton, J., DeWitt, D., Luo, Q., &. Lohman, G. (2001). On supporting containment queries in relational database management systems. *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data,* California.

Zibin, Y., & Gil, J. (2001, October 14-18). Efficient subtyping tests with PQ-encoding. *Proceedings of the 2001 ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Application,* Florida (pp. 96-107).

KEY TERMS

Branching: A branching is a subgraph of a directed graph, in which there are no cycles and the indegree of each node is 1 or 0.

Cyclic Graph: A cyclic graph is a directed graph that contains at least one cycle.

DAG: A DAG is a directed graph that does not contain a cycle.

Graph Encoding: Graph encoding is a method to assign the nodes of a directed graph a number or a bit string, which reflects some properties of that graph and can be used to facilitate computation.

Strongly Connected Component (SCC): An SCC is a subgraph of a directed graph, in which between each pair of nodes there exists a path.

Transitive Closure: The transitive closure of a directed graph *G* is a graph *G**, in which there is an edge from node *a* to node *b* if there exists a path from *a* to *b* in *G*.

Tree: A tree is a graph with a root, in which the indegree of each node is equal to 1.