Clustering a DAG for CAD Databases

JAY BANERJEE, MEMBER, IEEE, WON KIM, SUNG-JO KIM, AND JORGE F. GARZA

Abstract—**A DAG (directed acyclic graph) is an important data structure which requires efficient support in CAD databases. It typically arises from the design hierarchy, which describes complex designs in terms of subdesigns. Often, subdesigns are shared by more than one higher-level designs, and a set of design hierarchies thus forms a DAG. Evaluating an access request to a design (sub)hierarchy requires a forward or backward traversal of a DAG.**

There are three interesting methods of traversing a hierarchy or a DAG: the well-known depth-first and breadth-first traversals, and a hybrid of these, which we call children-depth-first traversal method. We organize the nodes of a hierarchy or a DAG into a clustered sequence of nodes which is (approximately) the sequence in which the nodes are visited in a given traversal method. This approach makes it possible to retrieve all descendants of a given node N in a single sequential fetch of the clustered sequence from secondary storage starting from the node N.

In this paper, we study the properties of the three types of clustered sequences of nodes for hierarchies and DAG's, and develop algorithms for generating the clustered sequences, retrieving the descendants of a given node, and inserting new nodes into existing clustered sequences of nodes which preserve their clustering properties. We also provide a performance comparison of the clustering sequences.

Index Terms—**Breadth-first traversal method, CAD databases, children-depth-first traversal method, clustering, DAG, depth-first traversal method, hierarchy, storage organization.**

I. INTRODUCTION

THE designers of complex engineering artifacts, such as a microprocessor chip, a large software system, a complex assembly of mechanical parts, etc., must resort to a *hierarchical decomposition methodology* [8] to keep the complexity of their designs humanly manageable. In a hierarchical decomposition methodology, the design of a complex artifact is described in terms of a design hierarchy where each node of the hierarchy represents the description of a component of the artifact, and the component is described in progressively greater detail along the hierarchy. Although a complex design can be completely described in terms of a design hierarchy, each node of a design hierarchy may belong to more than one design hierarchy. Thus the design artifacts, represented as nodes of design hierarchies in a CAD database, form a directed acyclic graph (DAG), rather than a set of independent hierarchies.

Manuscript received August 29, 1986; revised June 4, 1987.

J. Banerjee was with MCC, 3500 West Balcones Center Drive, Austin, TX, 78759. He is now with Unisys Corporation, P.O. Box 64942, St. Paul, MN 55164.

W. Kim and J. F. Garza are with MCC, 3500 West Balcones Center Drive, Austin, TX 78759.

S.-J. Kim is with the Department of Computer Sciences, University of Texas, Austin, TX 78712.

IEEE Log Number 8823670.

The designers or CAD tools (application programs) may issue requests against the database which will require the traversal of a design hierarchy downward or upward from a given node to retrieve all qualified descendants or ancestors of the node. Retrieval of the descendants of a node on a DAG requires a "downward" (forward) traversal of the DAG, while an "upward" (backward) traversal is required for the retrieval of the ancestors. In [9] we describe a number of typical retrieval requests against design DAG's expressed in a query language suitable for CAD databases.

In this paper, we will limit our discussions to downward traversals of hierarchies and DAG's. Our methodology will, however, apply to upward traversals equally well. In the case of a hierarchy, any given node has at most one parent, but in general many child nodes. As such, a downward traversal of a hierarchy touches many more nodes than an upward traversal. In the case of a DAG, any given node (except the root nodes) can have many parents and many children. However, we expect that a downward traversal of a DAG is much more frequent than an upward traversal.

Further, we will assume that a CAD database is disk-resident, and as such, the storage structures we will investigate in this paper will be designed to minimize disk I/O costs. There is a growing consensus in the CAD community that future CAD systems will be a distributed system consisting of intelligent workstations and a central database server. We accept the technology forecast that, as the cost of RAM continues to decrease, the size of the main memory in a workstation will grow. However, we also subscribe to the view that the size of the database that a designer will manipulate on a workstation will in general exceed the size of the virtual memory in the workstation.

As already discussed, CAD objects are modeled as DAG's. However, to the best of our knowledge, there are no efficient ways to organize DAG's in secondary storage. This motivated us to try to match the traversal patterns on DAG's in CAD databases to storage sequences of nodes, so that disk I/O costs are minimized. The *depth-first* and *breadth-first* traversal methods [7] are two well-known methods of downward-traversing a hierarchy (DAG). We will introduce a third method, we call *children-depth-first* traversal method, which combines the depth-first and breadth-first traversal methods. Each of these methods recursively traverses a hierarchy (DAG), visiting the nodes in some sequence. Our approach to minimizing the disk I/O cost in retrieving all qualifying descendants of a given

node is to organize the nodes of a hierarchy (DAG) in a sequence in which the nodes will be visited. In other words, the sequence in which nodes are organized in secondary storage, must correspond to the traversal sequence as closely as possible. We call such a sequence of nodes in secondary storage of a hierarchy or a DAG a *clustering sequence*.

The research contributions of this paper are as follows:

- It establishes properties of clustering sequences for the nodes of hierarchies and DAG's. Three clustering sequences are studied: depth-first, breadth-first, and children-depth-first.
- It presents algorithms for generating clustering sequences of hierarchies and DAG's, for expanding them with new nodes and edges, and for traversing these clustering sequences for finding all descendants of any arbitrary node. The algorithms preserve the properties of the clustering sequences.
- It provides the results of a simulation study on the performance of the three clustering sequences: against one another, and against nonclustered hierarchies and DAG's. We identify the circumstances under which each clustering sequence is superior to others.

The organization of the paper is as follows. In Section II we review some fundamental properties of the three methods of traversing a DAG. Section III identifies properties unique to each traversal method for hierarchies, and develops algorithms for inserting new nodes into an existing clustering sequence for a hierarchy. In Section IV we extend the hierarchy traversal methods to those for a DAG, and present algorithms for generating a clustering sequence for a DAG, retrieving the descendants of a given node of a DAG, and inserting new nodes into a DAG clustering sequence. Section V presents the results of our performance study of the three clustering methods for both hierarchies and DAG's. The paper is summarized in Section VI.

II. DAG-Traversal Methods and Clustering Sequences

In this section, we review the properties of three methods of traversing a hierarchy (DAG) to motivate our approach to organizing the nodes of a DAG into a clustering sequence. Two of these are the well-known *depth-first* and *breadth-first* traversal methods. The third, which we call a *children-depth-first* traversal method, combines the depth-first and breadth-first traversal methods.

A. Traversal Methods

In a depth-first traversal, a single branch of a DAG is pursued to the leaf level (or to a specified level), and only then are other branches explored. In a breadth-first traversal, a DAG is pursued one level at a time until the DAG is fully traversed (or until a specified level is reached). (We will use the term *level* of a node P in a DAG which is augmented with a virtual root node to mean the length of the longest path from the virtual root node G of the DAG to P.)

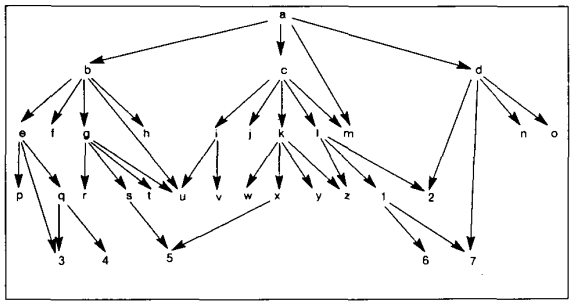


Fig. 1. An example of DAG.

We illustrate these traversal methods, with a DAG of depth 5 shown in Fig. 1. Assuming that the child nodes of a given node are traversed from left to right, the depth-first traversal method visits the nodes of the DAG, rooted at node a, in the following sequence.

a b e p 3 q 4 f g r s 5 t u h c

i v j k w x y z l 1 6 7 2 m d n o

If the DAG is traversed breadth-first, the following is the resulting sequence of nodes visited.

a b c m d e f g u h i j k l 2 7

n o p 3 q r s t v w x y z 1 4 5 6

The way in which these traversal methods visit the nodes may be characterized as follows.

- In a depth-first traversal of a DAG, once a node is visited, all previously unvisited descendants of the node are visited uninterrupted by any node that is not a descendant of the node. (In the case of hierarchies, all descendants of a given node are visited uninterrupted by any node that is not a descendant of the node.)
- In a breadth-first traversal of a DAG, the nth level descendants, for any n, of a given node P are visited consecutively, except for those nodes that have at least one parent that is neither an ancestor nor a descendant of P. (In the case of hierarchies, all nth level descendants of any given node P are visited consecutively.)

The depth-first traversal method is one in which, once a node is visited, all its children are visited immediately. The following procedural description defines the method more precisely.

```
PROCEDURE Children-Depth-First Traversal (P):
  IF node P was not previously visited THEN
  DO
    Visit node P;
        Visit ALL previously unvisited children of P;
      FOR EACH child C of P
          CALL Children-Depth-First (C);
  END;
  END PROCEDURE.
```

Using a children-depth-first traversal method, the nodes of the DAG in Fig. 1 will be visited in the following sequence.

a b c m d e f g u h p 3 q 4 r s

t 5 i j k l v w x y z 1 2 6 7 n o

The following summarizes the way in which the children-depth-first traversal method visits the nodes of a DAG.

- The child nodes of any node P are visited consecutively, except those that were visited before as children of some other nodes. This property is a weaker version of that of the breadth-first traversal method, where all nth level descendants of a node are visited consecutively.
- All previously unvisited descendants of a node are traversed consecutively. This is similar to the property of the depth-first traversal method, except that in a depth-first traversal visits to any child pair of a node are separated by the descendants of one of the child pair.

B. Clustering Sequences

The sequences of nodes of a DAG visited by the three DAG-traversal methods have two properties in common, which we can take advantage of in organizing the nodes of a DAG on secondary storage.

1) A node is visited before any of its descendants (which have not already been visited as descendants of some other node) are visited.

2) All descendants of a node (which have not already been visited as descendants of some other node) are visited in a nearly consecutive order.

If we organize the nodes of a DAG as a clustering sequence, that is, in a sequence in which they will be visited by a particular DAG-traversal method, the two properties above imply readily the following properties for the clustering sequence.

1) ABD (ancestors before descendants) Property: Any given node precedes *all* its descendants (which are not descendants of some other node).

2) CD (clustered descendants) Property: The descendants of any given node (which are not descendants of some other node) are *clustered* (physically close together).

A given DAG may be organized in one of three types of clustering sequence, corresponding to the three types of DAG-traversal methods we consider. For any type of clustering sequence, the ABD property guarantees that each physical block containing the nodes of a DAG will be retrieved only once; and the CD property implies that (close to) a minimum number of physical blocks will be retrieved. In other words, a clustering sequence for a DAG will minimize disk I/O's in traversing the DAG starting at any node. However, we emphasize that a clustering sequence is not in general identical to the sequence in which the nodes are visited in the corresponding traversal algorithm.

III. Clustering Sequences for Hierarchies

In this section, we first characterize more precisely the clustering properties of the three types of clustering sequence for hierarchies. We then present algorithms for inserting new nodes into a clustering sequence. The algorithms preserve all the clustering properties of each type of clustering sequence. Although they have some properties in common, the different types of clustering sequence do have characteristics unique to them. Algorithms for generating a clustering sequence from a given hierarchy of nodes, and for retrieving the descendants of a node from a clustering sequence are straightforward, and will not be shown here. In Section IV, we will generalize the discussions of this section to algorithms for DAG's. We refer the interested reader to [10] and [6] for discussions of depth-first clustering of hierarchies and breadth-first clustering of hierarchies, respectively.

A. Properties

The clustering property of a depth-first clustering sequence is as follows.

> Any given node P *and* all its descendants are clustered in a sequence uninterrupted by any node that is not a descendant of P.

A breadth-first clustering sequence has the following two clustering properties.

1) For any n, all nth level nodes of a hierarchy are clustered in a sequence uninterrupted by any node that belongs to any other level.

2) For any n, the nth level descendants of any given node P are clustered in a sequence uninterrupted by any node that is not an nth level descendant of P.

The following two properties characterize a children-depth-first clustering sequence.

1) All children of a given node P are clustered in an uninterrupted sequence.

2) All descendants of a given node P are clustered in a sequence uninterrupted by any node that is not a descendant of P.

We make the following assumptions about the representation of the nodes of a hierarchy. First, we assume that a hierarchy is represented as a table, named TREE, with one row for each node, as follows.

TREE (NodeName, NodeData, Children, Parent)

The NodeData column contains details about a node, for example, the description of the function of a part, the layout of a circuit, etc. The columns Children and Parent contain the names of the children and the parent, respectively, of the node identified by NodeName. The Parent column allows efficient access to direct ancestors of a given node.

Second, we assume that the values in the Children column are arranged in the order of their positions in the clustering sequence. Thus, for each node, it is meaningful to talk about a first child, next child, etc. The values in the Children and Parent columns are physical pointers to nodes.

B. Insertion Algorithms

A hierarchy of nodes may be modified, as new codes are inserted or existing nodes are deleted from the hierarchy. It is fairly straightforward to delete nodes from a hierarchy, since all that is required is, once a node is deleted, to make sure all its descendants are deleted as well. Further, inserting a node as the new root of a hierarchy is trivial, since the new node simply needs to be placed in front of the current root node in the clustering sequence. As such, in this section we develop algorithms only for inserting a new node as a child of an existing node of a clustering sequence. We also show that the algorithms ensure that insertions preserve the properties of the clustering sequences. The algorithms show how nodes are logically inserted into a clustering sequence; physical storage of clustering sequences is discussed in Section III-C.

1) Depth-First Insertion Algorithm: Insertion of a node C as a new child of a node P in a depth-first clustering sequence proceeds as follows.

1) Determine the address of the node P in the clustering sequence. (It is easily determined from an index on the NodeName colume of TREE.)

2) Place the new node C in the clustering sequence *immediately after* P. (C becomes the new first child of P.)

3) Update the Children column of P, to reflect the additional child of P.

As an example, the depth-first clustering sequence for the hierarchy of Fig. 2 is

a b f g c h j i k d e.

If we insert a node x as a new child of node c using the depth-first insertion algorithm, the resulting clustering sequence is

a b f g c **x** h j i k d e.

If we insert a node y as a child of node d, the new clustering sequence is

a b f g c x h j i k d **y** e.

Lemma 1 below shows that the above insertion strategy maintains the depth-first clustering property.

Lemma 1: The depth-first insertion algorithm preserves the depth-first clustering sequence.

Proof: Assume that, prior to the insertion of C, the clustering sequence is as follows: $\cdots P\ C1$ Desc-of-$C1$ $C2$ Desc-of-$C2$ $\cdots$ Cn Desc-of-Cn $\cdots$ where P has zero or more children $C1, C2, \cdots, Cn$, each of which is immediately followed by all its descendants. The new child C of P causes the clustering sequence to become $\cdots P\ C\ C1$ Desc-of-$C1$ $C2$ Desc-of-$C2$ $\cdots$ Cn Desc-of-Cn $\cdots$. We notice that P continues to be followed immediately by all its descendants. Further, each of its existing children $C1, C2, \cdots, Cn$, continues to be followed by all its descendants.

2) Breadth-First Insertion Algorithm: Insertion of a node C as a child of a node P in a breadth-first clustering sequence proceeds as follows.

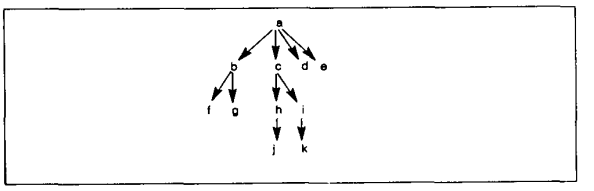


Fig. 2. An example hierarchy of nodes.

1) Scan the nodes in the forward direction, starting at P, until the first node Q is found that has at least one child.

2) If no such node Q is found, place C *at the end* of the clustering sequence. If Q is found, locate R, the first child of Q in the clustering sequence, and place C *immediately before* R.

The breadth-first clustering sequence corresponding to the hierarchy of Fig. 2 is

a b c d e f g h i j k.

If we insert a node x as a new child of c using the breadth-first insertion algorithm, the new clustering sequence is

a b c d e f g **x** h i j k.

The insertion of a node y as a child of d results in the sequence

a b c d e f g x h i **y** j k.

Lemmas 2 and 3 show that this simple algorithm preserves the two properties of a breadth-first clustering sequence.

Lemma 2: The breadth-first insertion algorithm guarantees that all nodes at a given level of a hierarchy are clustered in an uninterrupted sequence.

Proof: Suppose we are to insert a node C as a child of a node P which is at level n. Before the insertion, we know from the property of a breadth-first clustering sequence that all nodes at level $n + 1$ are in an uninterrupted sequence.

First we observe that, in the process of the forward scan from P, a node Q must be found which is either at level n or at level $n + 1$. This observation follows from the following facts: 1) Since the node at address P is at level n, all nodes following it must be at a level no less than n. 2) Q cannot be at any level n' larger than $n + 1$, because if there were such a node, then there would also be nodes at every level above (in particular, levels $n + 1, n + 2, \cdots, n' - 1$), and one of those would be encountered first.

If, in the process of the forward scan from P, a node Q is found at level n, then C is placed immediately before the first child of Q. Since that child must be at level $n + 1$, C is clustered with nodes at level $n + 1$. Alternatively, if the node Q is found to be at level $n + 1$, then Q must happen to be the very first node at level $n + 1$ that has a child. The first child of Q must, therefore, be the very first node at level $n + 2$. Since the new node C is placed

prior to that node, it is the last node at level $n + 1$, and, therefore, clustered with the nodes at level $n + 1$.

Lemma 3: The breadth-first insertion algorithm guarantees that the nth level descendants of a given node P are in a sequence uninterrupted by any nodes that are not the nth level descendants of node P.

Proof: Our proof of Lemma 3 consists of two parts. First, we will show that the breadth-first insertion algorithm ensures that the children of any given node are in a sequence uninterrupted by any other nodes. Second, we will show that if a node X precedes another node Y at one level, then the children of X precede those of Y.

These results will imply that the children of A, a predecessor sibling of X, will precede the children of X; and that the children of B, a successor sibling of X, will follow the children of X. In other words, the $(n + 1)$st level children of all nth level siblings (A, X, B) form a sequence of the children of A, X, and B, in that order, uninterrupted by any other nodes. By induction, all nth level descendants of a given node P form a clustered sequence uninterrupted by any other nodes.

The first part of the proof is trivial. Suppose we insert a node C as a child of P, which already has one or more children. The insertion algorithm places C immediately before the first child of P, obviously preserving the children of P in an uninterrupted sequence.

The second part of the proof requires a more careful consideration. Again suppose a node C is inserted as a child of P. If any node X precedes P, then every child of X must precede C. Since the node Q found in the forward scan from P follows X, its children must also follow those of X. C is placed immediately before these children; hence C must also follow the children of X.

We must also show that if P precedes any node Y, then C must precede the children of Y. Observe that, during the forward scan from P, Q is the first node found to have a child. Any node Y between P and Q has no children, and therefore does not violate the condition that C must precede the children of Y. C is placed before the children of Q, and before those of any node after Q. Thus, C precedes the children of every node that follows P.

3) Children-Depth-First Insertion Algorithm: The following algorithm ensures that insertion of a new node C as a child of a node P preserves the clustering properties of a children-depth-first clustering sequence.

1) If the node P has one or more children, place C *immediately before* the *first child* of P.

2) If P has no child, and P is the root of the hierarchy (i.e., P has no parent), place C *immediately after* P. (P's parent is found in the PARENT column of the record for the node P in the TREE table.)

3) If P has no child, but P is not the root of the hierarchy, determine its parent R. From the CHILDREN column of the record for R in the TREE table, determine L, the last child of R. Place C immediately after L.

We now illustrate the children-depth-first insertion algorithm. Again, using the hierarchy of Fig. 2, we see that the children-depth-first clustering sequence is

a b c d e f g h i j k.

If we use the children-depth-first insertion algorithm to insert a new node x as a child of the node c, the resulting clustering sequence is

a b c d e f g **x** h i j k.

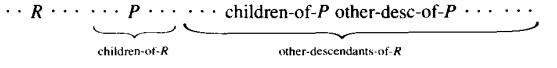
Insertion of another node y as a new child of the node d results in the sequence

a b c d e **y** f g h i j k.

Earlier, we defined two clustering properties of a children-depth-first clustering sequence. Lemmas 4 and 5 show that the children-depth-first insertion algorithm preserves these two properties.

Lemma 4: The children-depth-first insertion algorithm ensures that all children of any given node are placed somewhere after the node and they are in a sequence uninterrupted by any other nodes.

Proof: If P is not the root of the hierarchy, then assume that, prior to the insertion of C, the clustering sequence is as follows (where R is the parent of P):



If P has existing children, then C is placed immediately before the first child of P. Thus, all children of P follow P, and are uninterrupted by any other nodes. If P has no existing children, then C is placed immediately after the last child of R. Thus, all children of R remain together, and the only child of P follows P.

If P is the root of the hierarchy, and has existing children, then P is treated as shown above. If P is the root, and P has no existing child, then C is only the second node being placed in the clustering sequence (the first being the root itself). Therefore, the lemma holds for this case as well.

Lemma 5: The children-depth-first insertion algorithm maintains the clustered sequence of the descendants of any given node uninterrupted by any nodes that are not the descendants of the given node.

Proof: If P is not the root of the hierarchy, then refer to the clustering sequence shown in the proof of Lemma 4. Clearly, after the insertion of C, all descendants of R remain together. All descendants of every child of R, except P, are also left undisturbed. If P has existing descendants, the new child C immediately precedes all existing descendants of P. Thus all descendants of P remain together.

If P is the root of the hierarchy, the proof is as shown for Lemma 4.

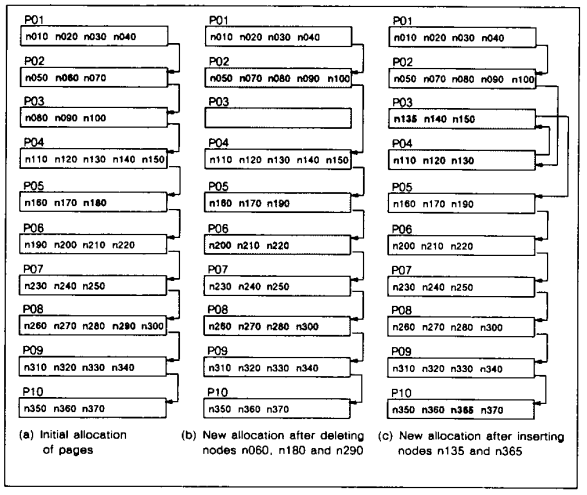


Fig. 3. Organization of a clustering sequence of nodes in secondary storage.

C. Physical Organization of the Clustering Sequence

Our proposal for organizing a clustering sequence on physical storage, and supporting insertions and deletions of nodes is based on the principles of B-tree organization [3], [4]. A clustering sequence of nodes will be stored in fixed-size pages, where each page is between 50 and 100 percent full. When a node is deleted from page P, but the deletion will leave the page at least 50 percent full, there is no problem. If, however, the deletion will make the page less than 50 percent full, the forward neighbor of P, say page Q, is examined. If the sum of the nodes in P and Q (after the deletion) will fit in a single page, they are placed in page P, and Q is dropped (returned to a free-page list). This process is similar to *page merging* in a B-tree. If it is not possible to place all nodes of P and Q in a single page, they are distributed evenly between the two pages. Fig. 3(a) shows a number of pages chained together, where each page may contain up to 5 records. Fig. 3(b) illustrates the effect of deleting some of these nodes. The nodes in pages $P05$ and $P06$ are redistributed, and the nodes in pages $P02$ and $P03$ are merged into a single page $P02$, thus causing $P03$ to become empty.

When a new node is inserted, its position within the clustering sequence is first determined. Once the page-id, say P, is determined for the new node, an attempt is made to store it in that page. If the page has enough space, the node is stored there. Otherwise, a new page Q is acquired from the free-page list, and the new node and the existing nodes in page P are distributed evenly between P and Q. This is similar to *page splitting* in a B-tree. Clearly, the two pages will now be at least 50 percent full. We illustrate the effect of node insertions in Fig. 3(c). As a result of inserting the node $n135$, the nodes in page $P04$ are distributed in two pages, $P04$ and a new page $P03$.

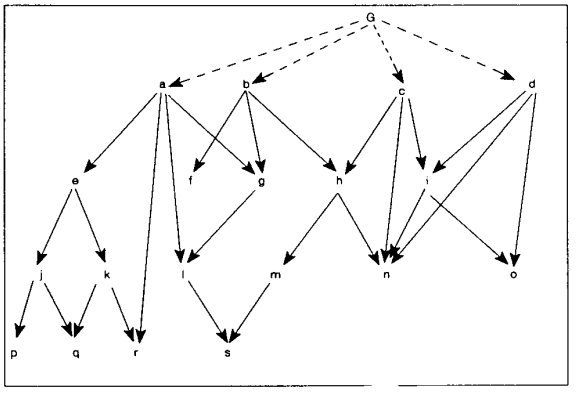


Fig. 4. An example of a rooted DAG (with a virtual root G).

IV. Clustering Sequences for DAG's

A DAG is a more general structure than a hierarchy, and, as such, the properties of the clustering sequences need to be examined anew for DAG's. We also need to develop algorithms for generating a clustering sequence for a DAG, for retrieving the descendants of a node in a DAG, and for inserting new nodes into a clustering sequence for a DAG. One important objective in developing the algorithms for DAG's is to ensure that they be applicable to hierarchies; otherwise, we would end up implementing one set of algorithms for hierarchies, and another for DAG's.

The fundamental insight underlying our algorithms for DAG's is to somehow *transform* a DAG into an equivalent hierarchy, and then develop retrieval and insertion algorithms for the resulting hierarchy. Since the algorithms will then be based on DAG-equivalent hierarchies, our algorithms for DAG's will be applicable to normal hierarchies, and, as such, will also be I/O efficient. Below we outline our approach to transforming a DAG to an equivalent hierarchy.

1) A DAG has in general multiple root nodes, and each nonroot node has multiple parents. To reduce a DAG to a hierarchy, we need first to augment a DAG to a *rooted DAG*, with a virtual root node G which has edges directed into the root nodes of the original DAG. Fig. 4 shows a rooted DAG.

2) The rooted DAG now has only one root, but each nonroot node can still have multiple parents. We can reduce the rooted DAG to a spanning tree [7]. A *spanning tree* for a DAG consists of all nodes of the DAG, but not all edges of the DAG. We can generate a spanning tree for a rooted DAG, such that each node retains at most one parent edge. Fig. 5 illustrates a spanning tree derived from the rooted DAG of Fig. 4.

3) All that remains now is to organize the spanning tree as a clustering sequence.

A clustering sequence for the spanning tree can only guarantee the ABD (ancestors before descendants) prop-

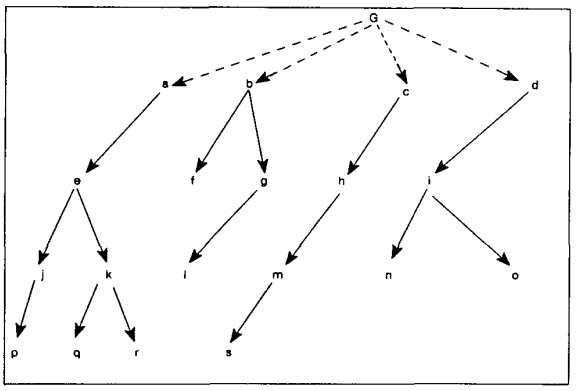


Fig. 5. A spanning tree for the DAG of Fig. 4.

erty only for the edges of the spanning tree, not for all edges of the DAG. This means that, although all descendants of a given node P in the spanning tree will follow P in the clustering sequence, those descendants of P not included in the spanning tree may not. For example, the descendants of the node b in the DAG of Fig. 4 include f, g, l, s, h, m, and n. However, in the spanning tree of Fig. 5, only the nodes f, g, and l are retained as the descendants of b. A clustering sequence based only on a consideration of the spanning tree will not guarantee that nodes h, m, s, and n will follow the node b.

However, as we will show in Section IV-B, we can generate a clustering sequence for the spanning tree, in which all descendants of a given node P that are included in the spanning tree *precede* those descendants of P that are not in the spanning tree. If we visit first the descendants of P that are included in the spanning tree and then the remaining descendants of P, we can retrieve all descendants of P in a single, sequential forward traversal of the clustering sequence.

More than one spanning tree may be found for a given DAG, and we must generate one that satisfies certain properties that facilitate efficient retrieval, which we will discuss next.

A. Properties of a Spanning Tree

We first define some terms related to a spanning tree for a DAG.

Definition: The children of a node P that are included in the spanning tree are called *direct children* of P; the remaining children of P, that are in the DAG but not in the spanning tree, are called *indirect children* of P.

For example, the node b in Fig. 4 has three children, f, g, and h. Fig. 5 shows that the nodes f and g are direct children of b, and h is an indirect child.

Definition: The only parent of P in the spanning tree is called its *direct parent*; the remaining parents of P are called its *indirect parents*.

For example, the node g is shown to have two parents, a and b, in Fig. 4. In the spanning tree of Fig. 5, we see that the direct parent of g is the node b; node a is g's indirect parent.

Definition: The descendants (or ancestors) of P in the spanning tree are called its *direct descendants* (*ancestors*); the remaining descendants (ancestors) of P are called its *indirect descendants* (*ancestors*).

Definition: The *level* of a node P in a rooted DAG is the length of the longest path from the virtual root node G to P. The level of a node is defined with respect to the DAG, and not the spanning tree.

We note that it is possible for an indirect child of a node P to be a direct descendant of P. For example, in Figs. 4 and 5, the node r is an indirect child of the node a, but it is a direct descendant of a.

We can now summarize the properties that a spanning tree must satisfy to qualify for a clustering sequence.

1) For depth-first clustering, any node is followed immediately by all its direct descendants, which in turn precede all indirect descendants.
2) For breadth-first clustering,
   a) all nodes at any level in the DAG are clustered,
   b) for each node P, the direct children are clustered, and they precede all indirect children; the level of the direct children follows that of node P.
   c) if any two nodes P and Q belong at the same level, and P precedes Q in the clustering sequence, then all direct children of P will precede all direct children of Q.
3) For children-depth-first clustering,
   a) all direct children of any node are clustered,
   b) all direct descendants of any node are clustered, and they precede all indirect descendants, and
   c) all direct children of any node P are followed immediately by all other direct descendants of P. Moreover, if any two nodes Q and R are direct siblings (i.e., they have the same direct parent), and Q precedes R in the clustering sequence, then all direct descendants of Q precede those of R.

We illustrate these properties in Fig. 6, where only the direct descendants of a given node are shown. We emphasize that the indirect descendants of a node always appear after the direct descendants.

Using Figs. 4 and 5, the reader may verify that each of the three clustering sequences satisfies its clustering properties.

The depth-first clustering sequences for the spanning tree of Fig. 5 is

G a e j p k q r b f g l

c h m s d i n o

The breadth-first clustering sequence is

G a b c d e f g h i j k

l m n o p q r s

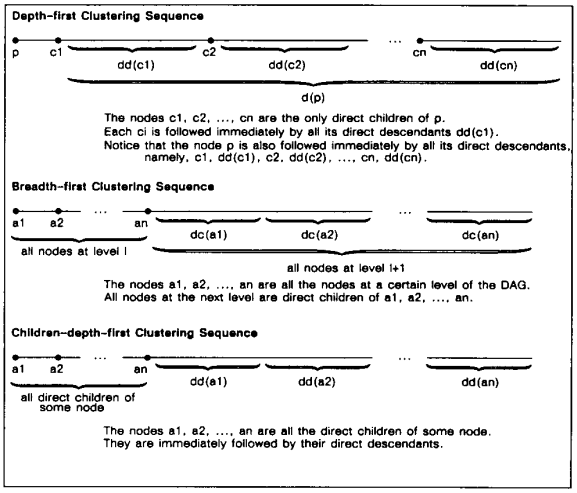


Fig. 6. Pictorial description of the properties of DAG clustering sequences.

The children-depth-first clustering is

G a b c d e j k p q r f

g l h m s i n o

B. Clustering Algorithms

We now present algorithms for generating a spanning tree of a DAG as a clustering sequence. We assume that the nodes of a given unclustered DAG are represented in a table

OLD-NODE (NodeName, NodeData, Children, Parents),

where each row corresponds to a node of the DAG. The clustering sequences generated by the algorithms will be a new table.

NODE (NodeName, NodeData, DirectChildren, DirectParent, IndirectChildren, IndirectParents). In the NODE table, the Children set of the OLD-NODE table is split into DirectChildren and IndirectChildren sets, and the Parents set is split into a single DirectParent and a set of IndirectParents. The rows of the NODE table are ordered in a clustering sequence.

The separation of the children of a given node into direct and indirect children is the essential difference between the clustering algorithms for the spanning tree of a DAG and the traversal algorithms on hierarchies. In each of the clustering algorithms, we will define a node Q to be a *direct child* of a node P (and conversely, P to be a *direct parent* of Q), if, among all parents of Q, P is the one with the largest position number in the clustering sequence (the NODE table). All other children of P in the original DAG (i.e., the OLD-NODE table) are *indirect children*; and all other parents of Q are *indirect parents*.

1) Depth-First Clustering: Depth-first clustering starts by inserting the virtual root node G into the clustering sequence (the NODE table), and then calling the recursive procedure Depth-First-Cluster to cluster all direct descendants of G. The direct children of G include all the original root nodes of the DAG.

1. *Initialize NODE with the row*
   (G, 'virtual root', NULL, NULL, NULL, NULL).
   / G is the virtual root node */*
2. *CALL Depth-First-Cluster(G).*

The procedure Depth-First-Cluster(P) appends the direct descendants of P to NODE, by recursively performing the following sequence of steps: 1) append a direct child of P, and 2) append the direct descendants of that child. The procedure, as also will be the case with the breadth-first and children-depth-first clustering algorithms, appends a node to the clustering sequence only when all its parents have been included in the clustering sequence.

```
PROCEDURE Depth-First-Cluster(P).
FOR EACH child C of P,
  IF C is not in NODE, AND
       IF all parents of C are in NODE THEN
  DO
    /* C is a Direct Child of P.        */
     Append C to the DirectChildren set of P.
     Append to NODE the row for C
          (C, 'C-Data', NULL, P, NULL, {Indirect
          Parents of C}).
    /* Indirect Parents of C are all parents of C in
       the OLD-NODE table, except P.      */
     FOR EACH IndirectParent Q of C,
          Append C to the IndirectChildren set of
          Q.
     CALL Depth-First-Cluster(C).
  END;
END PROCEDURE.
```

Lemma 6: The Depth-First Clustering Algorithm guarantees that all direct descendants of a node precede all indirect descendants of the node.

Proof: This is obvious.

2) Breadth-First Clustering: The procedure Breadth-First Cluster iterates over the number of levels in the DAG. On the first iteration, nodes at level 1 (the original roots) are added to the clustering sequence (the NODE table). At each subsequent iteration, all nodes at the next level are appended to the clustering sequence.

In the procedure, the variable OldList holds a list of pairs (P, C), for each P already in NODE, and each direct child C to be added at the next level of the DAG. The variable NewList holds a list of pairs (C, C'), where C' is a direct child of some node C in a pair (P, C) in OldList. On each iteration, the algorithm appends to the end of NODE a node C from OldList, determines the di-

rect children of C, and appends them to NewList. In the next iteration of the procedure, NewList is copied to OldList.

```
PROCEDURE Breadth-First-Cluster
Initialize NODE with the row
            (G, 'virtual root', NULL, NULL, NULL,
            NULL).
Set NewList to {(G, r)}, where r is a root of the original DAG.

WHILE NewList is not empty
  DO
    OldList = NewList.
    NewList = NULL.
    FOR EACH pair (P, C) in OldList
      DO
        /* C is a Direct Child of P, for all C and P
        in OldList.                                  */
         Append C to the DirectChildren set of P.
         Append to NODE the row for C
              (C, 'C-data', NULL, P, NULL, {In-
              direct Parents of C}).
        /* Indirect Parents of C are all parents of
        C in the OLD-NODE table, except P.           */

         FOR EACH IndirectParent Q of C,
           Append C to the IndirectChildren set of
           Q.

         FOR EACH child C' of C,
           IF all parents of C' are in NODE
              THEN Append (C, C') to NewList.

        /* C' is a Direct Child of C.                */
        /* NewList thus generated will contain only
        DirectChildren of C.                         */
        /* NewList is copied to OldList for the next
        iteration of this breadth-first clustering
        procedure.                                   */
        END;
  END;
END PROCEDURE.
```

Lemma 7: The Breadth-First Clustering Algorithm guarantees that all direct children of a node precede all indirect children of the node.

Proof: After the row for a node C is inserted into NODE, its direct children are immediately placed in NewList. Any indirect children of C will be entered later into NewList (perhaps even in another later iteration of "WHILE NewList is not empty"). Since rows are entered into NODE in the order they appear in OldList (which is a copy of NewList), it is clear that all direct children of C will precede all indirect children of C.

The key element of the algorithm is that it enters the nth level nodes (each C in the (P, C) pairs in OldList) *one at a time*, rather than all of them at once, into the clustering sequence.

Consider what would happen otherwise, that is, if the nth level nodes were entered simultaneously. Suppose an $(n-1)$st level node P has two children at the nth level of the DAG, $C1$ and $C2$, and $C3$ is a node with $C1$ and $C2$ as parents. Further, $C2$ has another child $C4$, whose only parent is $C2$. Suppose we place $C1$ and $C2$ in the clustering sequence at once, and then attempt to place the children of $C1$, followed by the children of $C2$. Since $C1$ and $C2$ are both in the clustering sequence, $C3$ will be included in NewList (therefore, in the clustering sequence) as a direct child of $C1$. This means that $C3$ will become an indirect child of $C2$. Since the algorithm places all direct children of $C2$ after direct children of $C1$, $C2$ now has an indirect child $C3$ preceding a direct child $C4$. That would be a violation of the breadth-first clustering property (Lemma 7).

3) Children-Depth-First Clustering: The children-depth-first clustering sequence is generated by first placing the virtual root G into the clustering sequence, and then calling the recursive procedure Children-Depth-First-Cluster to cluster the direct descendants of G. We will use an integer variable NextEmptyRow to hold the row number of the next available position in the clustering sequence.

```
1. Initialize NODE with the row
   (G, 'virtual root', NULL, NULL, NULL, NULL).
      /* Thus G occupies the first row of NODE */
2. Set NextEmptyRow to 2.
3. CALL Children-Depth-First-Cluster(G).
```

In the procedure Children-Depth-First-Cluster, n consecutive rows of NODE are first reserved for the n direct children of a node, starting at the row indicated by NextEmptyRow. However, the children are not inserted into the reserved rows all at once. Only one direct child C is entered at a time, followed immediately by the direct descendants of C starting at NextEmptyRow + n, by recursive calls to the procedure. Lemma 8 will show that this strategy is essential to ensure that all direct descendants of a node precede all its indirect descendants in a children-depth-first clustering sequence.

```
PROCEDURE Children-Depth-First-Cluster(P).
FOR EACH Child C of P,
  IF all parents of C are in NODE
     THEN Append C to the DirectChildren set of P

/* LocalRow is a local variable, while NextEmptyRow
is a global variable.                                */
LocalRow = NextEmptyRow.
NextEmptyRow = NextEmptyRow + number of
DirectChildren of P.

FOR EACH Direct Child C of P
     DO
     Insert into the LocalRow position of NODE
          the row (C, 'C-data', NULL, P, NULL, {In-
          direct Parents of C}).
     LocalRow = LocalRow + 1.
```

```
FOR EACH IndirectParent Q of C,
        Append C to the IndirectChildren set of Q.
    CALL Children-Depth-First-Cluster(C).
    END;
END PROCEDURE.
```

Lemma 8: The Children-Depth-First Clustering Algorithm guarantees that all direct descendants of a node precede all indirect descendants of the node.

Proof: The key element of the algorithm is that the direct children of a node are entered into the clustering sequence one at a time, followed immediately by its direct descendants.

Consider what would happen otherwise. Suppose the node P has two children, $C1$ and $C2$, and $C3$ is a node with $C1$ and $C2$ as parents. Further, $C2$ has another child $C4$ whose only parent is $C2$. Suppose we place $C1$ and $C2$ in the clustering sequence at once, and then attempt to place the descendants of $C1$, followed by the descendants of $C2$. Since $C1$ and $C2$ are both in the clustering sequence, $C3$ will be included in the clustering sequence as a direct child of $C1$. This means $C3$ will become an indirect child and also an indirect descendant of $C2$. Since the algorithm places all direct descendants of $C2$ after direct descendants of $C1$, $C2$ now has an indirect descendant $C3$ preceding a direct descendant, the direct child $C4$.

C. Retrieval Algorithms

A node in a DAG may have multiple parents. Therefore, retrieval algorithms must keep track of the nodes it has retrieved in order to prevent redundant retrieval of nodes which have multiple parents. This means that the algorithms require a potentially substantial internal storage space. Our depth-first and children-depth-first retrieval algorithms keep this internal storage space small. For the breadth-first retrieval algorithm, a large internal storage space appears to be unavoidable. All our algorithms take advantage of the properties of the clustering sequences in order to traverse a clustering sequence only in the forward direction, while visiting each relevant node only once.

1) Depth-First Retrieval: The principle underlying our algorithm for depth-first retrieval of all descendants of a given node N is rather simple. The algorithm first retrieves all direct descendants of the node N, and then recursively retrieves all direct descendants of each of the indirect children of N. The direct descendants are retrieved first since they precede all indirect descendants in a depth-first clustering sequence.

The algorithm makes use of an internal list L to keep track of the *indirect descendants* of the given node N. We emphasize here that direct descendants of a node never enter this list. The list L is kept sorted by the positions of the nodes in the clustering sequence. Initially, all indirect children of the node N are placed in L. Once all direct descendants of the node N have been retrieved, the first node F (with the lowest position) in L is retrieved next, together with all direct descendants of F. In the process, nodes are eliminated from L if they turn out to be direct descendants of F. The indirect descendants of F are placed in the list, and the process iterates until the list L is exhausted.

The algorithm makes use of a global variable LVN (Largest Visited Node) to hold the name of the node last retrieved. If we use a "$<$" (less than) relationship between two nodes P and Q to mean that P precedes Q in the clustering sequence, then LVN is less than the positions of any unvisited (unretrieved) nodes, and greater than or equal to the positions of any visited nodes. Further, LVN is less than the positions of all nodes in L, since L only contains the names of unvisited nodes.

Retrieval of all descendants of a given node N from a depth-first clustering sequence for a DAG proceeds as follows.

```
Initialize list L with the given node name N.
WHILE L is non-empty
  DO
    /* Let F be the first node in L.                    */
    /* L is ordered by node positions;                  */
    /* hence the first node has the smallest position */
    Retrieve F.
    Remove F from L.
    CALL Depth-First-DAG-Retrieve(F).
  END;
```

The procedure Depth-First-DAG-Retrieve(P) retrieves all direct descendants of P, and then places P's indirect children, if any, in the list L.

```
PROCEDURE Depth-First-DAG-Retrieve(P).
/* All Direct and Indirect Children of P are found in
the row for P in the NODE table.                      */
FOR EACH DirectChild C of P
  DO
    Retrieve C.
    LVN = position_of C     /* in the clustering se-
    quence  */
    IF C is the same as the first node in L
       THEN Remove C from L.
/* If C is in L as an indirect child of some previously
visited node, it is removed from L, since C is now being
visited as a direct descendant of P.                 */
    CALL Depth-First-DAG-Retrieve(C).
  END;

FOR EACH IndirectChild C' of P
    IF LVN < position_of C' AND IF C' is not in L
       THEN Add C' to L.
/* LVN ≥ position_of C' means that C', which is an
indirect child of P, happens to be a direct descendant
as well, and has already been retrieved as such.     */
END PROCEDURE.
```

2) Breadth-First Retrieval: The algorithm for breadth-first retrieval is rather straightforward. An internal list L keeps track of unvisited nodes that are the children of the nodes already visited. The list is ordered by the positions of the nodes in the clustering sequence.

```
PROCEDURE Breadth-First-DAG-Retrieve(N).
Initialize list L with the given node N.
WHILE L is non-empty
  DO
    Let P be the first node name in L.
/* L is ordered by node positions; hence the first node
has the lowest position                              */
    Retrieve P.
    Remove P from L.
    FOR EACH child C of P,
      /* regardless of whether C is a direct child or
      an indirect child of P.                       */
      IF C is not in L, THEN Add C to L.
  END;
END PROCEDURE.
```

3) Children-Depth-First Retrieval: A children-depth-first clustering sequence closely resembles a depth-first clustering sequence. The essential difference is that direct siblings (direct children of a common parent) occupy consecutive positions in a children-depth-first clustering sequence; while in a depth-first clustering sequence, two direct siblings are separated by the direct descendants of the first sibling. To retrieve a node N and all its descendants, the following steps need to be executed:

```
Initialize list L with the given node N.
WHILE L is non-empty
DO
/* Let F be the first node name in L.          */
/* L is ordered by node positions;             */
/* hence the first node has the lowest position */
  Retrieve F.
  Remove F from L.
  If position_of first node in L ≥ position_of any
  DirectChild of F
    THEN CALL Children-Depth-First-DAG-Re-
    trieve(F).
    ELSE FOR EACH child C of F /* regardless if
    Direct or Indirect Child */
        IF C is not in L THEN add C to L.
END;
```

The procedure Children-Depth-First-DAG-Retrieve(P) retrieves all direct children of P, and then all direct descendants of the direct children of P. Finally, it places the indirect children of P, if any, in the list L.

```
PROCEDURE Children-Depth-First-DAG-Re-
trieve(P).
FOR EACH DirectChild C of P
  DO
    Retrieve C.
    LVN = position_of C.
    If C is the same as the first node in L
    THEN Remove C from L.
/* If C is in L as an indirect child of some previously
visited node, it is removed from L, since C is now being
visited as a direct descendant of P.                */
  END;

FOR EACH DirectChild C of P
    CALL Children-Depth-First-DAG-Retrieve(C).

FOR EACH IndirectChild C' of P
    IF LVN < position_of C', AND IF C' is not in L
      THEN Add C' to L.
/* LVN ≥ position_of C' means that C', which is an
indirect child of P, happens to be a direct descendant
as well, and has already been retrieved as such.  */
END PROCEDURE.
```

D. Insertion Algorithms

In this section, we show how new leaf nodes and edges may be added to an existing clustering sequence for a DAG [1], [2]. The properties of the clustering sequence are preserved upon completion of the insertion.

1) Depth-First-DAG Insertion Algorithm: In the following algorithm a new leaf node C is added, together with a set E of edges into that node. The set E consists of the names of the parent nodes of C.

```
PROCEDURE Depth-First-DAG-Insert (C, E)
IF E is a null set, i.e., the new node has no incoming
edges,
  THEN append to the end of the NODE table the row
      (C, 'C-Data', NULL, NULL, NULL, NULL)
    and exit the procedure.
Determine the node P in E that has the largest position
in NODE.
Immediately after the row for P in the NODE table,
add the row
      (C, 'C-Data', NULL, P, NULL, E − {P})
    where P is made the DirectParent of C, and the
    remaining nodes in E are made IndirectParents
    of C.
Add C to the DirectChildren set of P.
Add C to the IndirectChildren set of every other node
in E.
END PROCEDURE.
```

To illustrate the algorithm, we consider the DAG of Fig. 4. Suppose that we are inserting a new leaf node t, which has parents b and h. Further suppose that, prior to the insertion of t, the clustering sequence is

G a e j p k q r b f g l

c h m s d i n o

Since the position of h is larger than that of b, the new node t is inserted immediately after h. Therefore, h is the direct parent of t, and b is an indirect parent of t. The clustering sequence now becomes

G a e j p k q r b f g l

c h t m s d i n o

Lemma 9: The Depth-First Insertion Algorithm preserves the depth-first DAG clustering sequence.

Proof: Since the new node C is placed after all its parents, it does not disturb the ABD property of the clustering sequence. The fact that the new node immediately follows its largest parent P, and is a direct descendant of P, implies that P continues to be followed immediately by all its direct descendants.

2) Breadth-First-DAG Insertion Algorithm: Using the following algorithm, a new leaf node C is added in breadth-first order, together with a set E of edges into that node.

PROCEDURE Breadth-First-DAG-Insert(C, E).
IF E is a null set,
THEN let P=G where G is the virtual root node of the DAG
ELSE determine the node P in E that has the largest position in the NODE table.
Scan the nodes in the forward direction, starting at P, until the first node Q is found that has at least one direct child.
Locate the first direct child of Q in the cluster sequence,
and immediately before the row for that child of Q in the NODE table
add the row (C, 'C-Data', NULL, P, NULL, E − {P})
where P is made the DirectParent of C,
and the remaining nodes in E are made the IndirectParents of C.
IF no such a node Q is found
THEN add the row (C, 'C-Data', NULL, P, NULL, E − {P})
at the end of the clustering sequence.
Add C to the DirectChildren set of P.
Add C to the IndirectChildren set of every other node in E.
END PROCEDURE.

With respect to Fig. 4, suppose we are inserting a new leaf node t which has parents b and h and that the position of h is larger than that of b. Before inserting t as a direct child of h, the clustering sequence is

G a b c d e f g h i j k

l m n o p q r s

After inserting t, the clustering sequence results in

G a b c d e f g h i j k

l t m n o p q r s

Lemma 10: The Breadth-First Insertion Algorithm preserves the breadth-first DAG clustering sequence.

Proof: Assume that a new leaf node C is being added, together with a set E of edges into that node. We are to insert the node C as a direct child of the node P at level n, where the node P in E has the largest position in NODE table. Before the insertion, the original clustering sequence satisfies the property of a breadth-first DAG clustering sequence. Starting at P, we look for the first node Q that has at least one direct child. Note that Q must be found either at level n or at level $n + 1$.

If the node Q is found at level n, then C is placed immediately before the first direct child of Q. Since that child must be at level $n + 1$, C is clustered with nodes at level $n + 1$. Further, since C is placed before any direct children of Q, all direct descendants still precede the indirect descendants at the same level. On the other hand, if there exists any previously existing indirect child (Pic) of P which precedes the first direct child (Qdc) of Q in NODE table, then after the insertion the property of the clustering sequence will be violated. We argue that it is impossible. If Pic's direct parent precedes P in NODE table, then P should be direct parent of Pic rather than indirect parent. If Pic's direct parent follows Q (Q itself can be Pic's direct parent), it violates the assumption that the original sequence satisfies DAG clustering sequence since Pic precedes Qdc.

Alternatively, if the node Q is found at level $n + 1$, then Q must happen to be the very first node at level $n + 1$ that has a direct child. The first direct child of Q must, therefore, be the very first direct node at level $n + 2$. Since the new node C is placed prior to that node, it is the last node at level $n + 1$; C is clustered with nodes at level $n + 1$ after the insertion. As in the previous case that Q is found at level n, we can prove that it is impossible a P's indirect child (if any) precedes a Q's direct child in NODE table; the direct descendants precede the indirect descendants at the same level.

If such a node Q cannot be found at any level, a new leaf node C should be placed as the last direct child in the clustering sequence. Further P must have no indirect children (if so, the node Q should be found at level n). Therefore the ABD property can be maintained when it is placed at the end of the clustering sequence.

Finally, we can easily show that after for the nodes X and Y at some level if X precedes Y in the clustering sequence before the insertion of C, then all direct descendants of X still precede those of Y after the insertion.

Consequently, the breadth-first-DAG insertion algorithm preserves the breadth-first-DAG clustering sequence.

3) Children-Depth-First-DAG Insertion Algorithm: Insertion of a new node C as a child of nodes in a set E of edges in a children-depth-first clustering sequence proceeds as follows.

PROCEDURE Children-Depth-First-DAG-Insert(C, E).
IF A is a null set,

THEN let P=G where G is the virtual root node of the DAG
ELSE determine the node P in E
that has the largest position in NODE table.
IF the node P has one or more direct children
THEN add the row (C, 'C-Data', NULL, P, NULL, E − {P}) immediately before the row for the first direct child of P.
IF P has no direct children and P=G
THEN add the row (C, 'C-Data', NULL, P, NULL, NULL) immediately after the row for P.
IF P has no direct children and P ≠ G
THEN DO
Scan the predecessor siblings of the node P in backward direction until the first node Q is found that has at least one direct child.
Locate the rightmost direct descendant of Q in the clustering sequence, and add the row (C, 'C-Data', NULL, P, NULL, E − {P}) immediately after the row for the rightmost direct descendant.
If no such node Q is found among the predecessor siblings, determine L, the last direct child of P's direct parent, and add the row (C, 'C-Data', NULL, P, NULL, E − {P}) immediately after L.
END
Add C to the DirectChildren set of P.
Add C to the IndirectChildren set of every other node in E.
END PROCEDURE.

The children-depth-first clustering sequence corresponding to the DAG of Fig. 4 is as follows.

G a b c d e j k p q r f

g l h m s i n o

We are inserting a node t as a new child of b and h using the above algorithm. Since the position of h is larger than that of b and h has a direct child m, the new node t is inserted immediately after h. Now, the new clustering sequence becomes

G a b c d e j k p q r f

g l h **t** m s i n o

Lemma 11: The Children-Depth-First insertion algorithm preserves the children-depth-first DAG clustering sequence.

Proof: Assume that a new leaf node C is being added, together with a set E of edges into that node. We are to insert the node C as a direct child of the node P at level n, where the node P in E has the largest position in the NODE table. Before the insertion, the original clustering sequence satisfies the property of a children-depth-first clustering sequence.

First, consider the case that P is not a root of the DAG. Then, prior to the insertion of C, the clustering sequence is as follows:

$$\cdots R \cdots \underbrace{P \cdots}_{\text{direct-children-of-}R} \underbrace{\text{direct-children-of-}P \ \text{other-desc-of-}P \cdots}_{\text{other-direct-descendants-of-}R}$$

where R is the direct parent of P.

If P has existing direct children, then C is placed immediately before the first direct child of P. Thus, all direct children of P follows P and are uninterrupted by any other nodes. After the insertion of C, all direct descendants of R remain together and all direct descendants of every direct child of R also remain together, since the node C is inserted as the first direct child of P whose position is the largest in E and as an indirect child of the remaining nodes in E as well. Next, assume that, after the insertion of C, the node C precedes a direct descendant (ldd) of one of its direct parents (l). It is impossible; otherwise, the original sequence cannot be children-depth-first clustering sequence, i.e., ldd should be a direct descendant of P rather than l. As a result, all direct descendants of any node still precede all indirect descendants after the insertion.

If P has no existing direct children, then C is placed immediately either after the rightmost direct descendants of its nearest predecessor slibling or after the last direct child of its direct parent. Thus, all direct descendants of P and its siblings remain together, and the only direct child of P follow P. In this case, all direct descendants of any node also precede all indirect descendants. It is inserted after all direct descendants of its indirect parents (if any) since P has the largest position in E in the clustering sequence and P precedes the new leaf node C. Therefore, the node C, even when inserted as an indirect child, follows the direct descendants of its indirect parents.

Next, consider the case that P is a root of the DAG. If P has existing direct children, then P can be treated as above. If P has no existing children at all, then C is only the second node being placed in the clustering sequence. Therefore the lemma holds trivially for this case as well.

Finally, we can easily prove that if any two nodes X and Y are direct sibling and X precedes Y before the insertion of C, then all direct descendants of X still precede those of Y after the insertion.

Consequently, the children-depth-first-DAG insertion algorithm preserves the children-depth-first clustering sequence.

V. Performance Analysis

In this section, we compare and analyze the performance of the retrieval methods for clustered DAG's and hierarchies. In our simulation study, we assumed that clustering sequences are stored in pages that contain a fixed number of records, as described in Section III-C. We assumed that a node requires about 20 to 50 bytes, and a page ranges in size from 4K to 20K bytes. Conse-

quently, we had 10, 100, or 1000 nodes per page. The measure of performance was the number of data pages fetched from disk.

We consider two types of access requests to hierarchies and DAG's, which we believe are the most important for CAD applications:

1) Retrieve all descendants of a specified node (e.g., fully flatten a CAD design hierarchy for a timing simulation).

2) Retrieve only the children of a specified node (e.g., expand a CAD design hierarchy one level at a time, for an interactive editing and design-rule check).

A. Hierarchy Clustering Sequences

We have conducted an extensive set of experiments for both hierarchies and DAG's. For hierarchies, we limited the experiments to *complete hierarchies* (hierarchies in which all nodes, except the leaves, have a fixed number of children). We did not investigate the effects of irregularities in the hierarchies; we used the experiments with hierarchies to gain some insight into the performance of DAG clustering sequences.

We used four types of complete hierarchies, with fan-outs of 2, 3, 4, and 5, respectively. The results were similar in every case; hence we present the results only for the case of fan-out 4. The hierarchies had 9 levels, with the root node at level 0, and leaf nodes at level 8. Thus the total number of leaf nodes was 4**8 = 65 536, and the total number of nodes in the hierarchy was 88 767.

Figs. 7(a) and 7(b) show the access cost ratios (ratios of page I/O's) against the start level number, which is the level number of a given node whose descendants are to be retrieved. Access cost ratios are shown for depth-first versus children-depth-first, and breadth-first versus children-depth-first.

Fig. 7(a) shows that depth-first clustering is the best when *all* descendants of a node are retrieved. However, depth-first requires at most one fewer I/O than children-depth-first. Breadth-first is clearly the worst, especially when the start node is chosen from one of the middle levels in the hierarchy.

Fig. 7(b) is for the situation when a given node and only its children are retrieved. Children-depth-first is usually better than depth-first, except when the start node is a level or two above the leaves. Breadth-first is better than children-depth-first when the start level is close to the root, and worse when the start level is below the middle levels of the hierarchy.

Variations in the page size also exhibited some interesting behavior. Larger page sizes favor breadth-first (relative to the other methods) when the start node is close to the root. This is because the first few levels of the breadth-first clustering sequence tend to occupy the same page. However, for such large page sizes, breadth-first performs relatively worse when the start level number is farther away from the root. This is because, as page size is

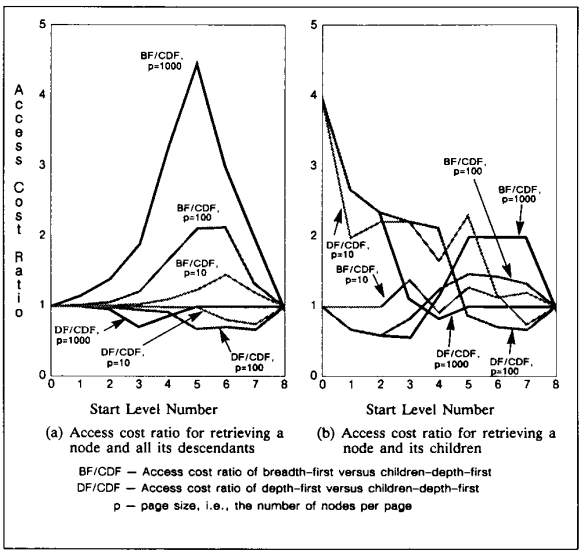


Fig. 7. Access cost ratio for retrieving nodes.

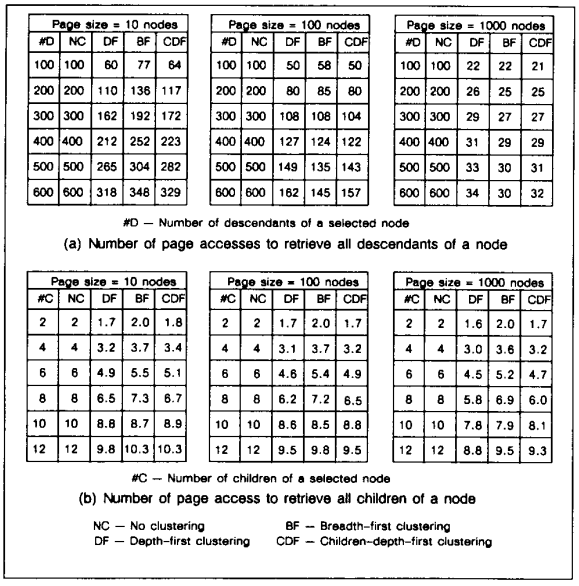
increased, the other two methods reduce their I/O costs more rapidly than does breadth-first, especially when the start level is below the middle levels of the hierarchy. In such situations, a node and its descendants at all levels tend to occupy the same page in a depth-first or children-depth-first clustering sequence.

We conclude that the children-depth-first clustering method is the best for hierarchies. In retrieving all descendants of a node, children-depth-first is outperformed by depth-first, but by at most a single I/O, in general a negligible fraction of the total number of pages accessed. Further, either of these methods is considerably more efficient than breadth-first. In retrieving the children of a given node, children-depth-first and breadth-first are almost equally good, and each outperforms depth-first. However, the breadth-first retrieval algorithm requires a large amount of internal storage. This drawback of breadth-first retrieval has also been observed in [5].

B. DAG Clustering Sequences

To measure the performance of the three clustering sequences for DAG's, we generated a random DAG with 50 000 nodes and 150 000 edges. Thus, the average fan-out and fan-in of each node was 3. We first clustered the DAG using the three algorithms, depth-first, breadth-first, and children-depth-first. We then applied the two types of accesses on 1000 randomly selected nodes.

In Fig. 8, we tabulate the results of the experiment. The first column of the tables in Fig. 8(a) lists the number of descendants of a selected node. The first column of the tables in Fig. 8(b) lists the number of children of a node (plus 1, since the node itself must be retrieved). The second column in each table is a rough estimate of the number of data pages to be accessed if nodes are stored without clustering. We assume that only a single page is



Page size = 10 nodes

#D	NC	DF	BF	CDF
100	100	60	77	64
200	200	110	136	117
300	300	162	192	172
400	400	212	252	223
500	500	265	304	282
600	600	318	348	329

Page size = 100 nodes

#D	NC	DF	BF	CDF
100	100	50	58	50
200	200	80	85	80
300	300	108	108	104
400	400	127	124	122
500	500	149	135	143
600	600	162	145	157

Page size = 1000 nodes

#D	NC	DF	BF	CDF
100	100	22	22	21
200	200	26	25	25
300	300	29	27	27
400	400	31	29	29
500	500	33	30	31
600	600	34	30	32

#D — Number of descendants of a selected node

(a) Number of page accesses to retrieve all descendants of a node

Page size = 10 nodes

#C	NC	DF	BF	CDF
2	2	1.7	2.0	1.8
4	4	3.2	3.7	3.4
6	6	4.9	5.5	5.1
8	8	6.5	7.3	6.7
10	10	8.8	8.7	8.9
12	12	9.8	10.3	10.3

Page size = 100 nodes

#C	NC	DF	BF	CDF
2	2	1.7	2.0	1.7
4	4	3.1	3.7	3.2
6	6	4.6	5.4	4.9
8	8	6.2	7.2	6.5
10	10	8.6	8.5	8.8
12	12	9.5	9.8	9.5

Page size = 1000 nodes

#C	NC	DF	BF	CDF
2	2	1.6	2.0	1.7
4	4	3.0	3.6	3.2
6	6	4.5	5.2	4.7
8	8	5.8	6.9	6.0
10	10	7.8	7.9	8.1
12	12	8.8	9.5	9.3

#C — Number of children of a selected node

(b) Number of page access to retrieve all children of a node

NC — No clustering
DF — Depth-first clustering
BF — Breadth-first clustering
CDF — Children-depth-first clustering

Fig. 8. Access costs for recursive queries.

buffered; hence if two consecutive nodes to be accessed are in different pages, two page accesses will be required. If no clustering is done, the estimated number of pages to be accessed is roughly the same as the total number of nodes to be retrieved. The third, fourth and fifth columns are the number of pages accessed when the three clustering sequences are used.

From the tables, we conclude that all three clustering sequences are a considerable improvement over non-clustered sequences. Depth-first is probably the best when only a small number of nodes are retrieved. The differences diminish when a much larger number of nodes are retrieved. Finally, page size has a marked effect on performance for each of the clustering sequences. When large numbers of descendants are to be retrieved, they are clustered in fewer and fewer pages, as the page size increases. Consequently, clustering becomes more and more important as pages become larger.

VI. Summary

A set of design objects in a CAD database forms a DAG. An important problem in a CAD database is to be able to retrieve all descendants of a given node of such a DAG with a minimum disk I/O cost. In this paper, we proposed an approach to organize the nodes of a DAG as a clustering sequence. In a clustering sequence, the nodes of a DAG are placed on secondary storage in a sequence which approximates the sequence in which the nodes are visited in a particular DAG-traversal method, and, more importantly, which ensures that all descendants of a node of a DAG are fetched in a single forward scan of the DAG.

We studied the properties of three DAG-traversal methods: the well-known depth-first and breadth-first traversal method, and a hybrid method, called the children-depth-first traversal method, that combines the essential characteristics of the depth-first and breadth-first traversal methods. These traversal methods share two properties which make a clustering sequence a storage organization for efficient retrieval of a DAG: a given node is visited before any descendants of the node, except those which are descendants of some other node, and the descendants of a node, except those which are descendants of some other node, are visited in a sequence uninterrupted by nodes which are not descendants of the node.

We presented simple algorithms for inserting new nodes into an existing clustering sequence for a hierarchy of nodes. We then extended the discussions to DAG's. The algorithms are based on the insight that we can transform a DAG into an equivalent hierarchy by augmenting the DAG with a virtual root node, then generating a spanning tree for the rooted-DAG, and then organizing the spanning tree as a clustering sequence. We provided algorithms for generating clustering sequences from the nodes of a given DAG, algorithms for retrieving the descendants of a given node, and algorithms for inserting new nodes into a DAG. Finally, we presented an analysis of the performance of the three clustering sequences for hierarchies and DAG's.

We suspect that deletion of arbitrary nodes and edges from a DAG may require extensive changes to the clustering sequences. Insertion of arbitrary new edges between existing nodes may also cause extensive changes. We have not developed algorithms for these operations, because they are not important for the types of data-intensive CAD applications we have examined. However, such applications may be important in other application domains involving DAG's, for example, recursive queries in logic-based AI applications. As such we offer it as a research issue to interested readers.

Acknowledgment

We thank the referees for carefully reading an earlier version of this paper and exposing a number of technical inaccuracies and points of potential misunderstanding.

References

[1] J. Banerjee, "A clustering algorithm based on recursive traversal patterns in graph models of CAD," MCC, Internal Rep., Oct. 1985.
[2] J. Banerjee and W. Kim, "Storage structures for evaluating recursion in CAD databases," MCC, Internal Rep., Oct. 1985.
[3] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," *Acta Inform.*, vol. 1, 1972.
[4] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surveys*, vol. 11, no. 2, pp. 121-137, June 1979.
[5] A. Guttman, "New features in a relational database system to support computer aided design," Ph.D. dissertation, Electron. Res. Lab., Univ. California, Berkeley, 1984.
[6] D. Hsiao, D. S. Kerr, and F. Ng, "DBC software requirements for supporting hierarchical databases," Ohio State Univ. Tech. Rep. OSU-CISRC-TR-77-1, Apr. 1977.
[7] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville, MD: Computer Science Press, 1978.
[8] R. Katz, *Information Management for Engineering Design* (Computer Science Survey Series). Heidelberg, Germany: Springer-Verlag, 1985.
[9] W. Kim, H.-T. Chou, and J. Banerjee, "Semantics and implementation of complex objects," in *Proc. Int. Conf. Data Engineering*, Los Angeles, CA, Feb. 1987.
[10] M. Schkolnick, "Clustering algorithm for hierarchical structures," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 27-44, Mar. 1977.

Jay Banerjee (S'77-M'79) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, the M.Tech. degree in computer science from the Indian Institute of Technology, Kanpur, and the Ph.D. degree in computer and information sciences from the Ohio State University, Columbus, in 1979.

He is presently the Director of Advanced Development in the Hardware-Independent Software Department at UNISYS. His areas of involvement include distributed processing, database systems, knowledge-based systems, and user interfaces. His previous work includes the development of a relational database system at Sperry, and an object-oriented database system at MCC, Austin, TX. He has published extensively in technical journals and conferences.

Dr. Banerjee is a member of the Association for Computing Machinery and the IEEE Computer Society.

Won Kim received the B.S. and M.S. degrees in physics from the Massachusetts Institute of Technology, Cambridge, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign. His Ph.D. dissertation was on query processing in relational database systems.

He is currently the Director of the Object-Oriented and Distributed Systems group within the Advanced Computer Architecture Program at MCC. The Object-Oriented Database System project in the group prototyped ORION, an object-oriented database system for AI, computer-aided design, and multimedia applications. Prior to joining MCC, he was a Research Staff member at IBM Almaden Research Center, where he participated in a number of database systems projects, including System D, highly available systems, and engineering design database systems. His research interests include object-oriented systems, distributed systems, multimedia systems, and knowledge-based applications.

Dr. Kim is currently an Associate Editor of *ACM Transactions on Database Systems*, and Chief Editor of *Data Engineering*, the quarterly bulletin of the IEEE Technical Committee on Data Engineering.

Sung-Jo Kim received the B.S. degree in applied mathematics from Seoul National University, Seoul, Korea, the M.S. degree in computer science from Korea Advanced Institute of Science, Seoul, Korea, and the Ph.D. degree in computer sciences from the University of Texas at Austin in 1988.

Previously, he was on the research staff of the Korea Institute of Defense Analysis. He was also an Instructor at the Department of Computer Sciences, Choong-Ang University, Seoul, Korea. He is currently an Associate Professor in the Department of Computer Science at Choong-Ang University. His current research interests include process mapping algorithms, multiprocessor operating systems, and multiprocessor architectures.

Mr. Kim is a student member of the Association for Computing Machinery and the IEEE Computer Society.

Jorge F. Garza received the B.S. degree in chemical and computer engineering from ITESM, Monterrey, Mexico, in 1980 and the M.S. degree in computer science from the University of Texas at Austin in 1985.

He worked for the computational center of ITESM as a software system specialist from 1980 to 1983. He is currently a member of the Technical Staff at the Microelectronics and Computer Technology Corporation, Austin, TX. His current research interests include physical database implementation, file management systems, distributed database systems, concurrency control, recovery, and page buffer management techniques.

Mr. Garza is a member of the Association for Computing Machinery and Phi Kappa Phi.