EFFICIENT MANAGEMENT OF TRANSITIVE RELATIONSHIPS IN LARGE DATA AND KNOWLEDGE BASES

Rakesh Agrawal *
Alexander Borgida **
H. V. Jagadish *

* AT&T Bell Laboratories, Murray Hill, New Jersey 07974
** Rutgers University, New Brunswick, New Jersey 08903

ABSTRACT

We argue that accessing the transitive closure of relationships is an important component of both databases and knowledge representation systems in Artificial Intelligence. The demands for efficient access and management of large relationships motivate the need for explicitly storing the transitive closure in a compressed and local way, while allowing updates to the base relation to be propagated incrementally. We present a transitive closure compression technique, based on labeling spanning trees with numeric intervals, and provide both analytical and empirical evidence of its efficacy, including a proof of optimality.

1. OVERVIEW

The significance of transitive relationships has long been acknowledged in varied fields. Recently, database researchers have recognized its utility in querying databases – in fact, it has been argued that transitivity is the dominant form of recursion that is of practical utility [2, 24, 27]. After briefly reviewing the motivation for managing large transitive relationships in databases, we present a second application area: the management and use of subclass (IS-A) hierarchies in knowledge representation systems. After considering the (conflicting) demands of efficient access, update, and storage, we argue that the usual techniques based on graph management are not adequate for managing and querying large transitive binary relationships, and propose a new technique for encoding such information. In Section 3, we present an algorithm for computing such an encoding, and show that it produces optimal results in a certain subclass of possible encodings – ones based on spanning trees for the graph of the relationship. We present both analytical and empirical evidence of its efficacy. Incremental update algorithms are briefly discussed in Section 4. In Section 5 we compare our approach with alternative techniques that have appeared in the literature. We conclude with some final remarks in Section 6. Some proofs and other details have been omitted for lack of space. These can be found in [4].

2. MOTIVATION

The computation of transitive relationships has been recognized as a sufficiently useful operation for it to have been included as an operator in a variety of query languages, based on both the relational and other data models [2, 11, 19, 21, 27]. A general technique for speeding up query processing in the presence of large amounts of data, even in the absence of such transitivity operator, is *view materialization* [7, 14]. Frequently accessed views are computed once and stored so that future queries can be answered directly, by look up, from such materialized views, instead of computing them on-the-fly. For the same reason, the problem of managing views which are the transitive closure of some relationship is of considerable interest, not withstanding recent progress on efficient algorithms for actually computing transitive relationships [1, 3, 6, 15, 16, 22, 29].

2.1 Transitivity in Knowledge Representation

Semantic networks and frame systems are among the most popular techniques for representing and reasoning with knowledge in Artificial Intelligence [9]. Systems based on these techniques allow concepts to be organized into *subclass hierarchies* (often known as "IS-A hierarchies"), with "inheritance" (the transitive traversal of such hierarchies) being a key component of their reasoning algorithms.[1]

More recently, knowledge representation systems such as KL-ONE [10] and its successors (e.g., [8, 20, 23, 26]) have

1. There are even proposals for knowledge representation systems in which *all* knowledge is represented as nodes connected by a variety of IS-A links[13]: Fred is hungry iff there is a path from the node labeled "Fred" to that labeled "hungry".

introduced compositional languages for defining concepts, where a concept is *subsumed* by another (belong under each other in the subclass hierarchy) by virtue of their definition: "all things whose children are doctors" is automatically more general than "all things whose children are eye-surgeons", if doctors subsume eye-surgeons. Computing the subsumption relationship between a new concept and previously known ones is the key inference made by such "terminologic logics"; in the process of making such computations, a frequent operation is finding out whether two previously known concepts are in the "subsumed by" relation. This relationship is therefore precomputed, cached as a hierarchy, and must be managed efficiently.

The space of concepts in a knowledge base can easily become quite large (an airplane, for example, may have close to 100,000 different *kinds* of parts – concepts), and must therefore be managed as a database, as in [30]. Questions about the transitive closure of the IS-A relationship, given their importance and frequency, must be answered by a technique more efficient than simple pointer chasing in the underlying data structure, the current approach.

2.2 Approaches to Management of Transitive Relationships

Binary relationships can be represented as directed graphs, which at least in the case of the above knowledge representation systems are acyclic. We have already argued that in both the case of large databases and knowledge bases it will be desirable to avoid the run-time computation of the transitive closure. This suggests that we materialize the transitive closure. However, the technique for storing materialized transitive closure must satisfy a number of conflicting requirements: storage cannot be used profligately, especially in main memory; look-up must be efficient, at least in the average case; in the case of large relations, the information will reside on secondary storage, and hence we need to minimize I/O traffic; updates (at least additions) to the base relation are not infrequent, so the incremental cost of adding new nodes and relationships should be less than recomputing the transitive closure.

These desiderata make unacceptable the obvious approaches for maintaining the transitive closure, such as linked lists or arrays of descendants, or 2-dimensional Boolean arrays, since the addition of all transitively derivable relationships can increase the number of edges in the graph from $O(n)$ to $O(n^2)$. Therefore, some compressed and local representation is needed. We now present such a compression scheme, which has been motivated by the technique for encoding trees, suggested by Schubert et. al. in [28].

3. THE COMPRESSION SCHEME

A binary relation, including a "source" field and "destination" field defined over the same domain, corresponds to a graph with a node for each distinct value of the source and destination fields and a directed arc for each tuple in the relation. The *successor list* of a node is the list of all nodes that are reachable from the specified node by traversing arcs of the graph, and the *immediate successor list* of a node is the list of all nodes to which the specified node has a direct arc. Similarly, the *predecessor list* of a node is the list of all nodes that can reach this node by traversing arcs of the graph, and the *immediate predecessor list* of a node is the list of all nodes that have a direct arc to the specified node. We concentrate on relations that correspond to acyclic graphs. However, the techniques presented in this paper can also be extended to cyclic graphs by collapsing strongly connected components into one node.

The generic compression scheme considered in this paper is *range compression*. The basic idea is to assign numbers to nodes so that instead of individually listing all nodes within a certain range of values in a successor list, the range is recorded. This form of compression is often used, for example, in citing references in technical literature. However, the way numbers appear in citations, this fortunate situation does not occur very often (see Section 2 of this paper, for instance). Our endeavor is to develop node numbering techniques that yield maximum compression. We do so by covering the graph with one or more spanning trees, and using the trees to generate node numbers.

We motivate our compression scheme by first considering the simple case when the original graph is a directed tree with arcs from parents to children, and then present the scheme for acyclic graphs.

3.1 Directed Trees

Number each node to reflect its relative position in a postorder traversal of the tree. The number of a node will be called its *postorder number*.

Now, assign to each node in the tree, an index consisting of the lowest postorder number among its descendents[2]. For simplicity, we assume that every node can reach itself, so that the index associated with a leaf node is the same as the postorder number of the node.

Lemma 1. Let the postorder number of node a be j, and let the index associated with a be i. There exists a direct path from node a to some other node b with the postorder number k *iff* $i \leq k < j$.

We thus have a compression scheme for trees that requires $O(n)$ storage, only a constant factor (twice) the storage for the tree itself, to store the transitive closure and can answer reachability queries with only one range comparison.

Consider, for example, the tree shown in Figure 3.1. The index and postorder number associated with nodes are shown as intervals. The first entry in an interval is the index and the second the postorder number. It can be immediately concluded that node b can reach h since the postorder number of h, which is 2, is contained in the interval [1,4] associated with b. However, node d cannot reach h, since the interval [6,7] associated with d does not contain 2.

2. For the case of trees, this scheme is identical to [28], wherein an interval consisting of the preorder number of the node and the highest preorder number among its descendents is associated with a node.

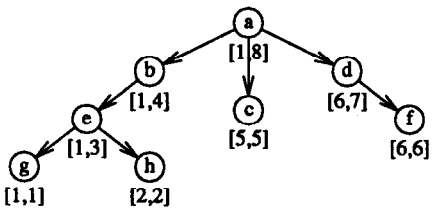

Figure 3.1. Compressed transitive closure for a tree

3.2 Directed Acyclic Graphs

We now generalize the above scheme for the case of a directed acyclic graph. We assume that the graph consists of only one connected component; disjoint components can be hooked together by creating a virtual root node.

The compression scheme works as follows:

1. Find a spanning tree T for the given graph G. We call T the *tree-cover* of G.
2. Assign postorder numbers and indices to the nodes of T as outlined in the Section 3.1. Thus, at the end of this step, an interval $[i,j]$ would be associated with each node, such that j is the postorder number of the node and i the lowest postorder number among its descendents.
3. Examine all the nodes of G in the reverse topological order. At each node p, do the following processing:
   — For every arc (p,q), add all the intervals associated with the node q to the intervals associated with the node p.
   — At the time of adding an interval to the interval set associated with a node, if one interval is subsumed by another, discard the subsumed interval. That is, if the two intervals $[i_1,i_2]$ and $[j_1,j_2]$ are such that $i_1 \leq j_1$ and $i_2 \geq j_2$, then discard $[j_1,j_2]$.

At each node now, there is a unique interval that encapsulates reachability information for the nodes that can be reached by following the tree arcs starting from this node.[3] We will refer to this interval as the *tree interval.* In addition, there are intervals associated with a node that encapsulate reachability information for the nodes that can be reached by following one or more non-tree arcs starting from this node. We will refer to such intervals as *non-tree intervals.*

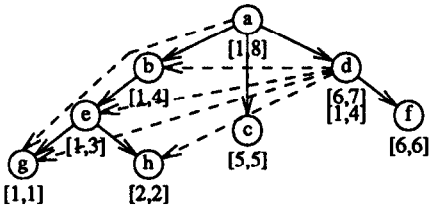

Figure 3.2. Compressed transitive closure for a DAG

3. A tree arc in G is an arc that is also in T. A non-tree arc in G is an arc that is not in T.

Consider, for example, the directed acyclic graph shown in Figure 3.2. Tree arcs have been indicated by solid arrows, and non-tree arcs by dashed arrows. The tree interval associated with node d is [6,7]. It inherits the non-tree interval [1,4] through the non-tree arc (d,b). It also inherits [1,3] through (d,e), [1,1] through (d,g), and [2,2] through (d,h), but these intervals are subsumed by [1,4]. Thus, only two intervals, [6,7] and [1,4], are associated with d. The non-tree interval [1,4] is inherited by a in turn, through the tree arc (a,d), but is subsumed by the tree interval [1,8] associated with a. Node a also inherits [1,1] through the non-tree arc (a,g), but this interval is also subsumed by the tree interval [1,8].

One can conclude that node d can reach h, since h's postorder number 2 is contained in one of the intervals ([1,4]) associated with d. However, d cannot reach a, since a's postorder number 8 is not contained in any of the intervals associated with d.

Optimum Tree-Cover

Nodes of a given graph can be covered by more than one spanning tree. However, all tree covers are not equally good.

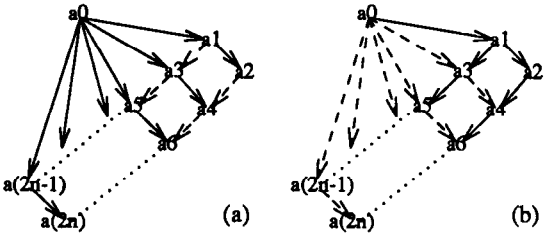

Figure 3.3. Two different tree-covers for the same graph

Consider, for example, the two different tree covers shown for the graph in Figure 3.3. The tree cover shown in Figure 3.3(a) requires $n^2 + n + 1$ intervals to represent the reachability information in the graph, whereas the optimal spanning tree cover, shown in Figure 3.3(b), requires only $3n$ intervals, reducing storage from being $O(n^2)$ to being $O(n)$. We now present an algorithm for finding the "optimum" tree-cover.

Let us first define optimality in this context. Each interval associated with a node causes one extra unit of storage in the compressed closure. It also causes additional effort at the time of look-up. Let each interval associated with a node place a unit weight on the node. The optimization problem then reduces to finding a tree-cover that results in minimum weight summed over all the nodes. The minimum weight obviously will result in minimum storage. If we assume uniform probability for reachability query between any two nodes, then this criterion is optimum from the average time viewpoint also.

If a node has two intervals associated such that one interval subsumes the other, then the subsumed interval can be deleted. That is, if a node has associated intervals $[i_1,j_1]$, $[i_2,j_2]$, with $i_1 \leq i_2 \leq j_2 \leq j_1$, then only the irst of the two intervals need be counted. Such interval subsumption is taken into account in what follows.

It may also happen that a node has associated with it two intervals $[i_1,j_1]$, $[i_2,j_2]$, with $i_2 = j_1+1$. In this case, one would think that a single interval $[i_1,j_2]$ would suffice. Such adjacent interval merging can cause problems as discussed later on. Two adjacent intervals count as two intervals for purposes of the following algorithm, lemmas, and theorem.

Here is the algorithm that obtains the optimum tree-cover for a given directed acyclic graph G:

Alg1 (Optimum tree-cover):

Topologically sort G;
Assume that all nodes with no predecessors are connected to a virtual level 0 root.

For every node j in G, in topological order, do:
  Repeat for each incoming arc pair (i_1,j), (i_2,j):
    if size(pred(i_1)) > size(pred(i_2)) then
      delete (i_2,j)
    else
      delete (i_1,j)
  For every i_k immediate predecessor of j
    pred(j) := $\{i_k\} \cup$ pred(i_k),

In the algorithm above, pred(j), is the set of all predecessors of node j, and is incrementally computed. There is a tree arc to each node from the immediate predecessor that has the largest pred() set.

Lemma 2. If a node j has two incoming arcs (i_1,j), (i_2,j), of which (i_1,j) is included in the tree-cover, then only those predecessors of i_1 that are not predecessors of i_2 will have to inherit the tree interval associated with j.

PROOF. Obviously, i_1 will have to inherit j's tree interval. Any predecessor of i_1 will also have to inherit this interval unless it is subsumed by some other interval associated with them. The way intervals are constructed, j's tree interval will be automatically subsumed by the tree interval associated with i_2. Any predecessor of i_2, therefore, does not have to inherit j's tree interval.

Lemma 3. If an interval $[i_1,i_2]$ subsumes another interval $[j_1,j_2]$, then there is path from i_2 to j_2 consisting solely of tree arcs.

PROOF. The interval $[i_1,i_2]$ represents a subtree of the spanning tree T such that its root is i_2 and i_1 is the smallest postorder number of any of its children. Similarly, the interval $[j_1,j_2]$ represents a subtree of the spanning tree T such that its root is j_2 and j_1 is the smallest postorder number of any of its children. Since $[i_1,i_2]$ subsumes $[j_1,j_2]$, we have $i_1 \le j_1$ and $i_2 \ge j_2$. Therefore, the subtree rooted at j_2 is contained in the subtree rooted at i_2, and there is a path consisting of tree arcs from i_2 to j_2.

Theorem 1. Alg1 gives the optimum tree-cover in the sense that the total number intervals associated with nodes (summed over all nodes) is minimum.

PROOF. Suppose there is another tree-cover T which is strictly better but does not follow from Alg 1. Topologically sort T. Starting from the lowest level (leaf-level), find the first level at which a node j exists that does not follow the rules of Alg1. That is, T contains an arc (i_1,j) but there is a predecessor i_2 of j such that size(pred(i_2)) > size(pred(i_1)). Transform T into $\tilde{T}$ such that the arc (i_1,j) is deleted and arc (i_2,j) is added. This change will cause a change in the number of intervals associated with the predecessors of j.

Let us represent by I_1 the set consisting of i_1 and its predecessors in G not also predecessors of i_2, and by I_2 the set consisting of i_2 and its predecessors in G who are also not the predecessors of i_1. By Lemma 2, the tree interval associated with j will now be inherited by all nodes in I_1 as a non-tree interval, but will no longer have to be inherited by nodes in I_2. Since size(pred(i_2)) > size(pred(i_1)), in the absence of interval subsumption, this change causes a net reduction in the total cardinality of interval sets associated with nodes.

Let us now consider the effect of interval subsumption. Two things can go wrong in the presence of interval subsumption:

First, the number of intervals associated with nodes in I_2 may not decrease since the tree interval of j which was inherited due to the non-tree arc (i_2,j) was already subsumed by some interval $[u,v]$ associated with some node $\tilde{i}_2$ in I_2 (see Figure 3.4). For this to happen, we claim that $\tilde{i}_2$ must also be a predecessor of i_1 in T and hence cannot belong to I_2. By Lemma 3, there must be a path consisting solely of tree arcs from v to j, and this path goes through the node i_1 as (i_1,j) is the tree arc coming into j in T. If v is same as $\tilde{i}_2$, we are done. Otherwise, there must be a path consisting of tree and non-tree arcs from $\tilde{i}_2$ to v along which the interval $[u,v]$ was inherited as non-tree interval. In that case also, $\tilde{i}_2$ is a predecessor of i_1.

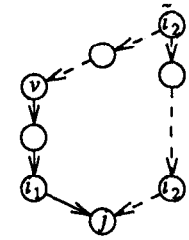

Figure 3.4. $\tilde{i}_2$ cannot belong to I_2

Second, the number of intervals associated with nodes in I_1 may not increase as the intervals inherited from j are subsumed by the intervals already present with nodes in I_1. First of all, the non-tree intervals inherited from j are not changed since by the very nature of the compression scheme, non-tree intervals are inherited along both tree and non-tree arcs. The only remaining question is whether the tree interval associated with j can be subsumed by an interval associated with a node $\tilde{i}_1$ in I_1 in T (see Figure 3.5). The answer again is no, since in that case the subtree rooted at j must be contained in the subtree rooted at $\tilde{i}_1$ or a node reachable from $\tilde{i}_1$ and there would be a path consisting solely of tree arcs from this node to i_2. Thus, $\tilde{i}_1$ would also be a predecessor of i_2.

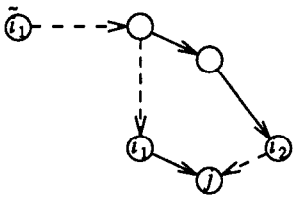

Figure 3.5. $\tilde{i}_1$ cannot belong to I_1

The above transformation is recursively applied to all nodes in the graph till T is transformed into a tree cover that is at least as good as T and follows the rules of Alg1. Hence Proved.

Note that the complexity of computing the compressed transitive closure of a graph is the same as the computation of its transitive closure. However, compression is a one-time activity, and once the compressed closure has been obtained, it can be repeatedly used to efficiently answer queries.

Storage Requirement

Alg1 minimizes the storage required for storing the compressed closure. We now examine how much storage is required for the compressed closure if the optimum tree cover generated by Alg1 is used for compression. There is a tree interval associated with every node in the graph. The number of non-tree intervals associated with a graph is given by the following lemma:

Lemma 4. The number of non-tree intervals associated with a node i equals the cardinality of the node set N_i, where N_i consists of all nodes j such that

i. there is a path from i to j containing one or more non-tree arcs, and
ii. there is no $k \in N_i$ such that there is path consisting solely of tree arcs from k to j.

PROOF. In absence of interval subsumption, a path from i to j containing one or more non-tree arcs propagates j's tree interval to i. We need not consider the non-tree intervals associated with j, since j must have inherited this interval from some other node l along a path from j to l that contains one or more non-tree arcs, but then this is a subpath of the path from i to l.

If there is a path from a node k to j consisting solely of tree arcs, then the tree interval of j is subsumed by the tree interval associated with k. Since k is in N_i, the tree interval of k is inherited by i and will subsume the tree interval of j. Hence proved.

The total storage requirement can now be obtained by summing the number of non-tree intervals over all the nodes, and depends on the nature of the graph. Obtaining a closed form formula for the total storage requirement in terms of suitably defined graph characteristics is an interesting open problem. In the worst case, the storage required for the compressed closure can be $O(n^2)$, as in the case of a bipartite graph shown in Figure 3.6.

One interval will be associated with each of the m bottom nodes and one of the top nodes. $m+1$ intervals are associated with each of the remaining $n-m-1$ top nodes. The total number of intervals required, therefore, is $(m+1)(n-m)$, which

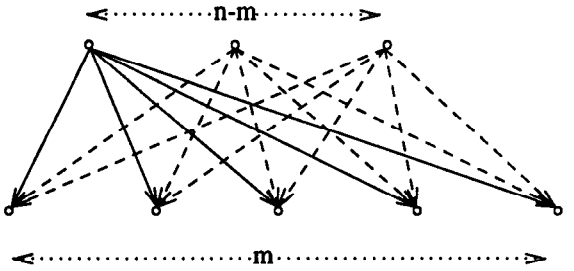

Figure 3.6. A bipartite graph with a large compressed closure

has a maximum value of $(n+1)^2/4$ for $n = 2m+1$.

The worst cases seem to occur when a large number of nodes have the same set of immediate successors. However, in such cases, a single common node can be created as an intermediary as shown in Figure 3.7, and the same information can be stored in just $(m+2) + 2(n-m-1) = 2n-m$ intervals, which is $O(n)$.

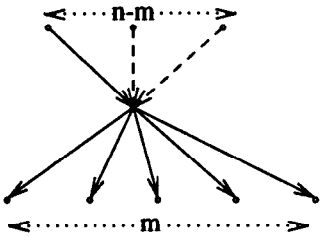

Figure 3.7. A similar bipartite graph with a small compressed closure

The creation of such "meaningful" bundles of objects is the whole essence of constructing inheritance hierarchies, and we do not expect worst case graphs to arise in the applications envisaged. Experience with Lassie, a classification-based software retrieval system [12], supports this belief.

We empirically investigate the average case behavior in Section 3.3. Before that, we look at some improvements that are possible.

Improvements

The number of intervals associated with a node can be reduced by merging adjacent intervals, that is, if the two intervals $[i_1,i_2]$ and $[j_1,j_2]$ are such that $j_1 = i_2 + 1$, then create one $[i_1,j_2]$ corresponding to these two intervals. It now becomes possible to generate overlapping intervals: merge two intervals, $[i_1,i_2]$ and $[j_1,j_2]$ into $[i_1,j_2]$, if $i_1 \le j_1 \le i_2 \le j_2$.

Such merging can sometimes result in significant reduction in storage requirement. For example, in Figure 3.6, the m non-tree intervals associated with $n-m-1$ top nodes can be collapsed into a single interval, bringing the total number of intervals required down to $(m+1) + 2(n-m-1) = 2n-m-1$, which is again $O(n)$.

Unfortunately, merging of adjacent intervals is order-dependent. Consider, for example, the following two graphs that are structurally equivalent, but lead to different compressions depending on how the children are ordered:

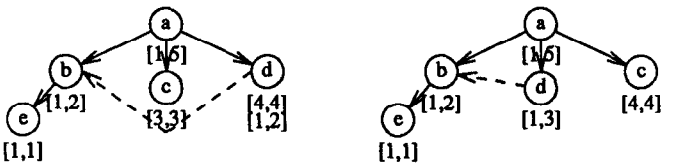

Figure 3.8. Order dependence in adjacent interval merging

No interval is merged in the first case. However, if the order of c and d were interchanged, the non-tree interval [1,2] inherited through the non-tree arc (d,b) can be merged with the tree interval [3,3] of the node. Finding an optimum ordering of node numbers to maximize the benefits of interval merging appears to be a combinatorial problem. We have omitted the merging of the intervals in Alg1 to be able to present a class of labelings for which Alg1 is optimal. In actual use, one may obtain tree cover using Alg1 and then merge adjacent intervals whenever encountered.

3.3 Performance Evaluation

In this section we present the results of simulation experiments performed to evaluate empirically the effectiveness of the proposed compression technique. The effectiveness has been measured in terms of the storage required for storing the transitive closure with and without compression. The total storage required was computed as the number of successors at each node for the original and transitive closure graphs. For the compressed transitive closure, in the simplest scheme, one has to store both end-points for every range interval. One may do better, for example, by storing the ranges separately and pointers to ranges at the nodes. We have computed the storage required for the compressed closure as twice the number of intervals required at each node to obtain baseline performance measure for the compression scheme.

Following [1], synthetic graphs were used as data sets in performance experiments. Two primary parameters define a database that can be represented as a graph: the average *degree* of a node and the *number of nodes*. We considered datasets with the average out-degree specified. The rationale for the choice of these parameters has been discussed at length in [1]. Experiments were run for various combinations of degrees and sizes, but we present here only the representative trends.

Figure 3.9 shows the storage requirement for the transitive closure of a random 1000 node graph as the degree of the graph is varied. The storage requirement is plotted as a multiple of the size of the original graph. Initially, as the degree of the graph is increased from 1 to 2 to 3, the size of the transitive closure increases rapidly as new paths are found to nodes that previously were unreachable. (A degree 1 graph is likely to consist of several unconnected components). Further increases in the degree do not increase the size of the closure as much, since most of the nodes are already reachable and addition of more arcs only establishes alternate paths to already reachable nodes. We found that of the 495,000 possible arcs in a 1000 node acyclic graph, 442,000 were already present in the closure of graph of degree 4. The size of the graph itself increases linearly with degree. Beyond a

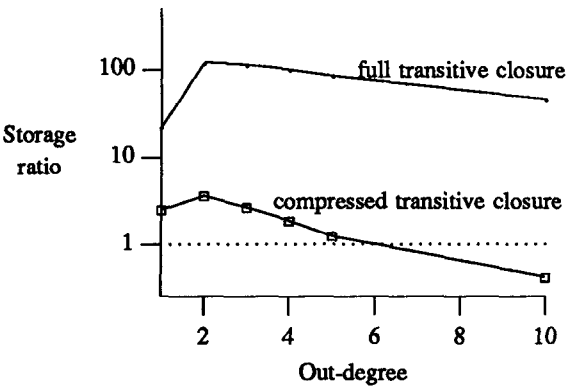

Figure 3.9. Storage required for a 1000 node graph as a function of average degree

certain degree, the size of the closure increases slower than the size of the graph, and a dip is observed in the relative size of the closure.

The size of the compressed closure also rises at first, though less so than the size of the full transitive closure. As more and more arcs are added, however, a deeper, less branchy tree can be found to cover the graph. Most successors of a node can be reached solely through tree arcs. Paths through non-tree arcs result in intervals that are mostly subsumed. Consequently, the size of the compressed closure actually begins to decrease as the degree of the graph is increased. This decrease contrasts with the increasing size of the original graph. Eventually the size of the compressed closure becomes even less than the size of the original graph itself.

That the compressed closure requires less storage than even the original graph may appear surprising at first. A graph of high degree has many "redundant" arcs whose removal does not affect the reachability information in the graph. The compressed closure avoids the extra storage required for these redundant arcs, resulting in less storage overall.

It can be argued that when the transitive closure includes most arcs in the graph, one should store the inverse, storing tuples only for source-destination pairs between which a path cannot be found in the graph. In the case of a directed acyclic graph, the maximum number of arcs in the graph is exactly half the total possible. When we say that most of the possible arcs have been includeed, we mean that the total number of arcs in the closure is close to $\frac{1}{2}n^2$. If a topological ordering of the graph is stored as well, then one can use the topological ordering to identify the $\frac{1}{2}n^2$ arcs that are possible according to this ordering. However, such a scheme makes incremental updates more complex as the topological sort may also have to be incrementally updated. In spite of this practical difficulty with inverse closures, we measured the size of the inverse closure with respect to a particular topological sort and obtained the results shown in Figure 3.10.

The size of the inverse closure falls rapidly as the degree of the graph is increased, and relative to the size of the graph, it falls even more rapidly. However, the size of the compressed closure stays well below that of the inverse closure, and decreases at a rate comparable to the inverse closure for high

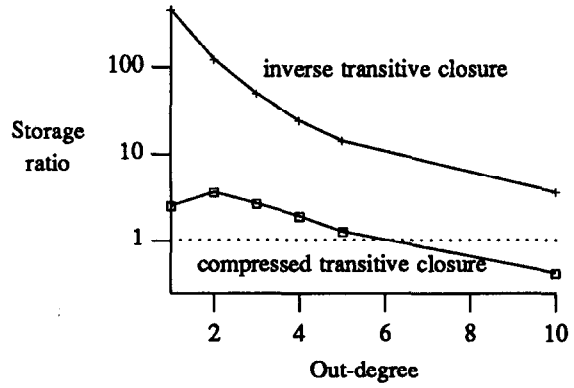

Figure 3.10. Storage required for a 1000 node graph as a function of average degree

degrees.

Figures 3.9 and 3.10 showed that the transitive closure of a graph can be compressed to a small fraction of its size by using the techniques presented in this paper. Figure 3.11 measures this compression for graphs with different numbers of nodes while the degree is kept fixed at 2. Storage requirement is again plotted as a multiple of the storage required for the original relation. Figure 3.11 clearly shows that, for these random graphs, the size of the compressed closure increases slower than the size of the full closure as the size of the graph is increased, giving better compression for larger graphs.

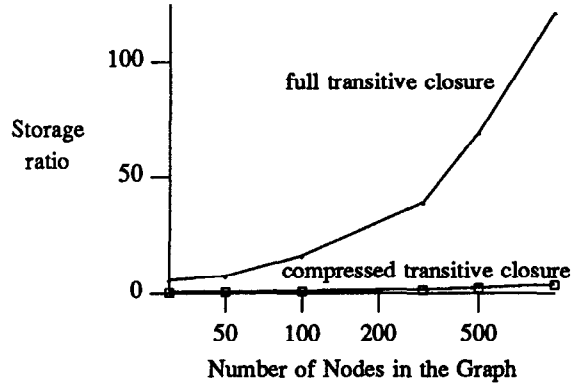

Figure 3.11. Storage required for a degree 2 graph as a function of number of nodes

We also performed a sensitivity experiment in which we generated all possible directed acyclic graphs of 8 nodes and computed the size of compressed closure in number of intervals. The result in Figure 3.12 demonstrates the infrequency of worst-case graphs.

The above experiments were performed without adjacent interval merging. We finally performed experiments in all cases to assess the benefits of interval merging. We found the additional compression obtained was rather small, usually less than 5%.

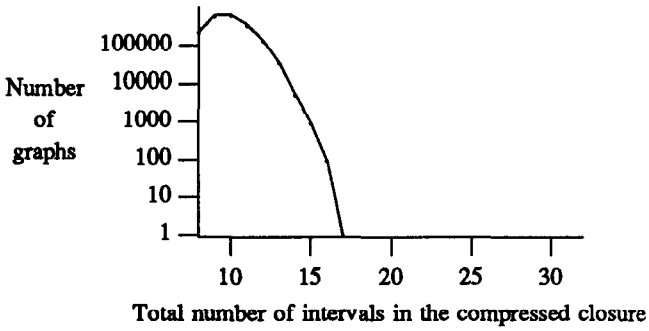

Figure 3.12. Frequency distribution of total number of intervals in the compressed closure of all possible 8-node acyclic graphs

4. INCREMENTAL UPDATES[4]

While assigning postorder numbers to nodes, it is not necessary to choose contiguous numbers; one can leave *gaps* between numbers and the compression scheme presented in Section 3 would still work correctly. The initial gap could be determined by dividing the range of integers that can be accommodated in one word by the number of nodes in the graph[5]. Figure 4.1 shows the intervals assigned to nodes in the graph given in Figure 3.2, where the gap between numbers is now 10 rather than 1.

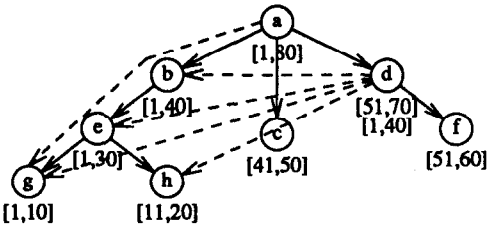

Figure 4.1. Compressed transitive closure with gaps in postorder numbers

The incremental update algorithms presented below exploit gaps in the postorder numbers assigned to nodes. In the following discussion, we assume that all the postorder numbers currently in use are maintained in a sorted list L.

4.1 Additions

Addition of a tree arc

Consider a new node j connected by a new arc (i,j) to an existing node i. Let the postorder number of i be n_2 and that of its immediate child with smallest postorder number be n_1 (n_1 is taken to be one less than the lower range of the interval associated with i, if i is a leaf node). Find the two postorder numbers between n_1 and n_2 that have already been assigned and have the largest difference. Let these be $\tilde{n}_1$ and $\tilde{n}_2$.

4. Only a brief sketch of the ideas is presented here. Further details may be found in [4].

5. Alternatively, one could use real numbers instead of integers.

Assign to j the postorder number $n = (n_1+n_2)/2$ and the interval $[\tilde{n}_1+1, n]$. Also, add n to L. No update is required in the intervals associated with any other node of the graph.

For example, in the graph of Fig. 4.1, the addition of node x and the tree arc (b,x) results in the postorder number 35 and the interval [31,35] to be assigned to node x. Similarly, the addition of node y and the tree arc (c,y) results in the postorder number 45 and the interval [41,45] to be assigned to y. No change is required in any other part of the graph.

The case where the newly added node j is connected to more than one existing node is treated as an addition of a tree arc followed by an addition of a non-tree arc.

Addition of a non-tree arc

If a non-tree arc (i,j) is added from the node i to node j, the intervals associated with j will have to be added to node i and all of its predecessors.

If the list of immediate predecessors is also maintained with each node, this propagation can be performed quite efficiently. Moreover, at the time of adding an interval to a node, if the new interval is subsumed by an interval already associated with the node, this interval need not be added. If no new interval is added to a node, the effect need not be propagated to the predecessors of this node.

If the predecessor list is not available with the nodes, then a scan over all the nodes would be necessary. At every node n, it must be checked if the node i can be reached from n; if yes, the intervals associated with j must be added to n.

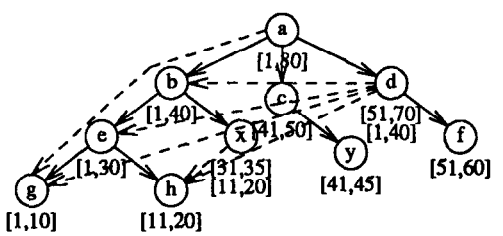

Figure 4.2. Compressed transitive closure after additions

The addition of the non-tree arc (x,h) in our running example will cause the interval [11,20] associated with h to be inherited by x and the predecessors of x. However, [11,20] is subsumed by the interval [1,4] associated with b and hence no new interval is added to b, a or d. The resulting compressed storage structure is shown in Fig. 4.2.

A further optimization is possible that can avoid propagation of updates to predecessors for certain kinds of additions. In the case of concept hierarchies in AI systems, when a new node is added and connected to existing nodes, the reachability set of the existing nodes is unchanged (except that some nodes may now reach this new node also). Such updates frequently take place while "refining" a hierarchy. An example of such an update in Figure 4.2 would be the creation of node z and arcs (e,z), (x,z), and (z,h). To handle such updates, one can provide an additional gap beyond the postorder number in the tree interval associated with a node. Thus, h's interval could have been made [11,25], with the understanding that nodes numbered 21 through 25 are not reachable from h. x will now inherit the interval [11,25] instead of [11,20] from h. Now when z is added, and if it is assigned a postorder number between 21 and 25, no update is required in both of its predecessors e and x, making hierarchy refinement a constant time operation.

What if empty numbers run out

It is possible that when a new node j and the tree arc (i,j) is being added, there is no empty postorder number available for j. However, nodes can be renumbered to create a gap so that j can be assigned a postorder number. Simply go left (or right whichever is shorter in the number of intermediate nodes) of the postorder number of i in L, and find the first hole. Suitably renumber all the intermediate numbers. Record this set of intermediate numbers as a set S of tuples of the form *<oldnum, newnum>*. Now make a scan over all the nodes of the graph. If any *oldnum* appears in any of the intervals, replace *oldnum* by *newnum*.

4.2 Deletions

Deletion of a tree arc

Assume that the deleted arc is (i,j) and l is currently the largest postorder number in L. Take the subtree rooted at j and make it a child of the virtual root. Renumber the nodes in the subtree assigning them numbers $> l$. (The postorder number of the virtual root is fixed at $+\infty$). Update tree intervals associated with the nodes in this subtree, but preserve non-tree intervals.

In the rest of the graph, modify any non-tree intervals that have old postorder numbers with new postorder numbers. Furthermore, if any of the tree predecessors of j had a non-tree arc coming into node k of the subtree rooted at j, they should now inherit the intervals associated with k.

Deletion of a non-tree arc

There is no change to the spanning tree of the graph. Perform a traversal of all the nodes in the reverse topological order, recomputing the non-tree intervals.

Deletion has special properties in AI concept hierarchies — nodes are "deleted" to be ignored, but the subset relationships between remaining nodes is unchanged, and no update is required to the compressed closure.

Finally, note that the incremental update algorithms presented in this section do not preserve the optimality of the tree-cover, and it may be prudent to develop a new tree-cover after sufficient update activity.

5. RELATED WORK

A technique for encoding reachability information in a hierarchy (tree) was proposed Schubert et. al. in [28], wherein an interval consisting of the preorder number of the node and the highest preorder number among its descendents is associated with a node. The same technique was also proposed, independently, by O'Keefe in [25], also to apply only in the case of a tree. For the simple case of tree, our technique is identical to this technique. Schubert *et al*

generalized their scheme somewhat to work for the case of overlapping hierarchies (*not* general directed acyclic graphs). Each hierarchy is treated independently and nodes are assigned intervals separately for each hierarchy. Thus, each node is assigned as many intervals as the number of hierarchies, and intervals associated with a node are differentiated by tagging them with the corresponding hierarchy identifiers. Hierarchies are taken as given: the decomposition of a graph into hierarchies is not addressed. Incremental updates were also not considered.

A transitive closure compression technique based on chain decomposition of graphs was proposed in [18]. Each node is indexed with a chain number, and its sequence number in the chain. At each node, one need store only the earliest node in a chain (the one with the lowest sequence number) that can be reached from it, and deduce that later nodes in the chain are reachable[6].

Theorem 2. For any graph G, its transitive closure can be compressed using postorder numbers on a tree cover to require storage less than or equal to the storage required by the best chain compression possible without chain reduction.

In other words, given any good chain cover, without reduction, we can do at least as well with our scheme. (See [4] for proof). On the other hand, there clearly are cases where a tree cover does significantly better than a chain cover. Consider, for example, a tree. As we saw in Section 3, $O(n)$ storage suffices to maintain closure in an n node tree. Significantly greater storage would be required by any chain compression technique.

Tree-like data structures that have a low amortized cost for incremental updates of transitive closure have been developed in [17]. However, this scheme is not targetted towards compression and requires more storage than the complete transitive closure.

Recently, a technique has been proposed to compute the greatest lower bound (and least upper bound) in a lattice efficiently [5]. While compressed transitive closure techniques, such as those presented in this paper, can be used for these lattice problems to advantage, further study is required to determine whether any of the ideas in [5] can be applied to our problem.

6. CONCLUSION

We presented a technique for compressing the transitive closure of a binary relation. The basic idea underlying this technique is to cover the graph with a spanning tree, use the position of a node in the tree to assign it a number, and then use these node numbers in storing successor lists at each node, keeping only the end-points of a range when all intervening nodes are also successors. Our method for obtaining the spanning tree is optimal in that it minimizes the total storage required to store the compressed closure under certain reasonable conditions. We empirically demonstrated the effectiveness of the proposed compression technique on a variety of graphs, and that in some cases, the storage required for the compressed closure can become even less than the storage required for the original relation. We also presented incremental algorithms that allow updates to the base relation to propagate to the materialized closure without having to recompute the whole closure.

The techniques presented in this paper may provide the basic building blocks for efficient implementation of future deductive and knowledge base systems. With the compressed closure, answering a transitive closure query in a deductive database system reduces to a lookup instead of a graph traversal. Indeed, we are planning to incorporate these techniques in prototype systems based on α-extended relational algebra [2]. These techniques are also useful for efficient propagation of inherited values and properties, and will be central to an implementation of the INCINERATE data model [19], in which the concept of inheritance is introduced in the relational data model. Similarly, we can use these compression techniques for the computation of subsumption, disjointness, least common ancestors, and other properties in frame-based knowledge representation systems. CLASSIC [8], a new knowledge representation system being developed in our laboratory, has separated the maintenance of subclass relationships into an abstract data type that maintains the IS-A graph and encapsulates the technique for managing this data structure efficiently. We plan to use the techniques presented in this paper for this purpose.

6. A chain reduction technique was also proposed in [18] that leaves some nodes uncovered by chains as a means of achieving further compression. We do not consider the additional compression offered by chain reduction in Theorem 2.

REFERENCES

[1] R. Agrawal and H. V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 255-266.

[2] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590. Also in *IEEE Trans. Software Eng.* 14, 7 (July 1988), 879-885.

[3] R. Agrawal and H. V. Jagadish, "Multiprocessor Transitive Closure Algorithms", *Proc. Int'l Symp. Databases in Parallel and Distributed Systems*, Austin, Texas, Dec. 1988, 56-66.

[4] R. Agrawal, A. Borgida and H. V. Jagadish, "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases", AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, 1989..

[5] H. Ait-Kaci, R. Boyer, P. Lincoln and R. Nasr, "Efficient Implementation of Lattice Operations", *ACM*

Transactions on Programmng Languages and Systems 11, 1 (Jan. 1989), 115-146.

[6] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations", Tech. Rept. DB-004-85, MCC, Austin, Texas, 1985.

[7] J. A. Blakeley, P. A. Larson and F. W. Tompa, "Efficiently Updating Materialized Views", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 61-71.

[8] A. Borgida, R. J. Brachman, D. L. McGuinness and L. A. Resnick, "CLASSIC: A Structural Data Model for Objects", *Proc. ACM-SIGMOD 1989 Int'l Conf. on Management of Data*, Portland, OR, May-June 1989.

[9] R. J. Brachman and H. J. Levesque, (ed.), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985.

[10] R. J. Brachman and J. G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science 9*, 2 (April-June 1985), .

[11] I. F. Cruz, A. O. Mendelzon and P. T. Wood, "A Graphical Query Language Supporting Recursion", *Proc. ACM SIGMOD Conf. on Management of Data*, 1986, 16-52.

[12] P. T. Devanbu, P. G. Selfridge, B. W. Ballard and R. J. Brachman, "Steps Toward a Knowledge-Based Software Information System", AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, 1989..

[13] S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, Cambridge, MA, 1979.

[14] E. Hanson, "A Performance Analysis of View Materialization Strategies", *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Fransisco, California, May 1987, 440-453.

[15] Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators", *Proc. 12th Int'l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 403-411.

[16] Y. E. Ioannidis and R. Ramakrishnan, "An Efficient Transitive Closure Algorithm", *Proc. 14th Int'l Conf. Very Large Data Bases*, Los Angeles, California, Aug.-Sept. 1988.

[17] G. F. Italiano, "Amortized Efficiency of a Path Retrieval Data Structure", *Theoretical Computer Science 48*, (1986), 273-281.

[18] H. V. Jagadish, "A Compressed Transitive Closure Technique for Efficient Fixed-Point Query Processing", *Proc. 2nd Int'l Conf. Expert Database Systems*, Tysons Corner, Virginia, April 1988.

[19] H. V. Jagadish, "Incorporating Hierarchy in a Relational Model of Data", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[20] T. S. Kaczmarek, R. Bates and G. Robins, "Recent Developments in NIKL", *Proc. AAAI-86,*, Philadelphia, PA, 1986, 978-985.

[21] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro and M. Stonebraker, "Heuristic Search in Data Base Systems", *Proc. 1st Int'l Workshop Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984, 96-107.

[22] H. Lu, "New Strategies for Computing the Transitive Closure of a Database Relation", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987.

[23] K. von Luck, B. Nebel, C. Peltason and A. Schmiedel, "The Anatomy of the BACK System", Kunstliche Intelligenz und Textverstehen - Report 41, Technical University of Berlin, Jan. 1987.

[24] E. Neuhold and M. Stonebraker, "Future Directions in DBMS Research", Tech. Rep.-88-001, Int'l Computer Science Inst., Berkeley, California, May 1988.

[25] R. A. O'Keefe, "A New Data Structure for Type Trees", in *ECAI-84: Advances in Artificial Intelligence*, T O'Shea (ed.), Elsevier Science Publishers, 1984.

[26] P. F. Patel-Schneider, "Small can be Beautiful in Knowledge Representation", *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, Colorado, Dec. 1984.

[27] A. Rosenthal, S. Heiler, U. Dayal and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 166-176.

[28] L. K. Schubert, M. A. Papalaskaris and J. Taugher, "Determining Type, Part, Color, and Time Relationships", *IEEE Computer 16*, 10 (Oct. 1983), 53-60.

[29] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices", *Proc. 1st Int'l Conf. Expert Database Systems*, Charleston, South Carolina, April 1986, 197-208.

[30] I. M. Walter, P. C. Lockemann and H. Nagel, "Database Support for Knowledge-Based Image Evaluation", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 3-11.