EFFICIENT SEARCH IN VERY LARGE DATABASES

Rakesh Agrawal
H. V. Jagadish

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We consider the problem of performing efficient search in a large database system. We present a novel data structuring technique and show how a branch and bound search algorithm can use the proposed data organization to prune the search space. Simulation results confirm that, using these techniques, a search can be expedited significantly without incurring a large storage penalty. As a side benefit, it is possible to organize the search to obtain successive approximations to the desired solution with considerable reduction in total search.

1. INTRODUCTION

The capability to process recursive queries is likely to be an essential feature in the next generation of database systems, and considerable research has recently been devoted to devising techniques for processing recursion. Extremal path problems [9] constitute a large and useful subclass of recursive queries and arise in several practical applications [3, 8, 16]. An extremal path problem on a graph involves the identification of a path between a pair of nodes in the graph that has an extreme value (highest or lowest on some precedence ordering) for its label, or the calculation of the value of such an extremal label. The label for a path is computed by applying a specified concatenation function to the labels of the arcs (or sub-paths) constituting the path. In addition, there may be constraints on nodes and/or arcs that may or may not be included in the desired path. Examples of such problems include the problem of finding the cheapest flight between two cities, the problem of finding the critical path in a project network, the problem of finding the most reliable path in a communication network, etc. Other examples of such problems appear in [3, 5, 8, 9, 14, 16].

The nature of extremal path problems is such that their solution often requires a search over a sizable data space. In this paper, we investigate how such a search can be performed efficiently over a very large database. We discuss the issues of both data organization and how the search can use this data organization effectively. To keep the discussion concrete, we shall use the shortest path problem as the running paradigm. However, later in the paper, we shall show how our approach extends naturally to other path problems.

Our overall approach is to *partially* precompute some information and then to use it at run time to prune the search space. We must hasten to add that we are considering really large databases, such as those with topographical map data. By an analysis similar to [13], one can estimate that simply to store a small 100 mile by 100 mile map discretized at 100 foot intervals, one requires about 2.4 Giga bytes of storage. From this, one can get an idea of the size of data involved when larger maps are considered. The size of path information in a transitive closure is considerably larger than the original relation. When data is of this magnitude, precomputing and storing path information, even after using the encoding and compression techniques that are proposed in [1, 12], would be infeasible. We are, therefore, proposing a new data organization technique in this paper, and we show how this data organization allows us to derive successively tighter bounds that must be satisfied by a point that is to be opened[1] during the search process. These bounds can cut down tremendously the number of data points explored by a search algorithm, such as Dijkstra's algorithm [10] for finding the shortest path between two points.

The organization of the rest of the paper is as follows. In Section 2, we present our data organization scheme based on the concept of domains, and show how it can be used for pruning the search without a large storage overhead. We also discuss how domains can be created in the first place. This basic two-level structure is extended to a multi-level structure in Section 3. In Section 4, we present simulation results that support our analysis developed in the previous sections regarding the effectiveness of our techniques. Some generalizations and related issues have been discussed in Section 5. We present our conclusions in Section 6.

1. We are using the phrase "open" to represent the set of possible next moves as used in the AI literature on search algorithms (see [7], for example). If a node under consideration is opened, then we shall, perhaps at a later time, investigate the possibility of the desired shortest path going through that node. If the bounds indicate that the desired shortest path cannot pass through the node, it is not opened.

2. DOMAIN ENCODING

In the state-space search paradigm [7], in order to find the solution of a problem, a search algorithm starts from one or more initial states and finds paths to the goal states. In absence of any criteria for determining whether an intermediate node should be explored (opened), the number of nodes explored before arriving at a solution is likely to be prohibitively large. We provide a bounding procedure to cut down on the number of intermediate nodes that are explored before the final solution is obtained. New algorithms can be designed using this bounding procedure, or it can be incorporated in any branch and bound search algorithm to improve its performance.

Our bounding procedure is based on partially precomputing some information, and then using it for developing successively tighter bounds for opening an intermediate node. If a node does not satisfy the bounds, it need not be opened. In this section, we describe what information is precomputed and how it is used to obtain the bounds. We also analyze the storage overhead due to this precomputed information and the extent to which the search space may be pruned by our bounding procedure.

2.1 Data Organization

Given a graph consisting of nodes, arcs between the nodes, and labels on these arcs (representing distance or some other appropriate quantity), divide the nodes into sets called *domains*, such that there exists a path from each node in a domain D to every other node in D. Each domain has a distinguished point called the *center of domain* or simply *center*. The *radius* of a domain D is the shortest distance between the center and a node in D that is farthest from the center. (If the distances to and from the center are different, the larger of the two gives the radius). We discuss how to form domains in Section 2.6. For now let us assume that such domains have somehow been created.

The shortest distance between all domain centers is precomputed and compressed using the techniques described in [1, 12]. Efficient techniques, such as those described in [4, 5], may be used for computing the shortest distances. In addition, the shortest distance between each node and its domain center (and vice versa, if different) is precomputed and stored.

2.2 A Lower Bound

Given the data organization described above, we will first derive a lower bound on the distance between two points that belong to different domains. This lower bound will be used in deriving the upper bound on the distance through a point which is being considered to be opened. If the distance through a candidate point is larger than the upper bound, this point is not opened.

Lemma 2.1. *Let D_1 and D_2 be two distinct domains with centers c_1 and c_2. Let $p_1 \in D_1$ and $p_2 \in D_2$. Then, the shortest distance from p_1 to p_2*

$$p_1p_2 \geq c_1c_2 - c_1p_1 - p_2c_2$$

where c_1c_2 is the shortest distance from c_1 to c_2, c_1p_1 is the shortest distance from c_1 to p_1, and p_2c_2 is the shortest distance from p_2 to c_2.[2]

PROOF. The lower bound on the distance from p_1 to p_2 is derived by considering the bound on the distance from c_1 to c_2. Since c_1c_2 is the shortest distance from c_1 to c_2, any alternate path from c_1 to c_2 is at least as big. We, therefore, have (see Figure 2.1)

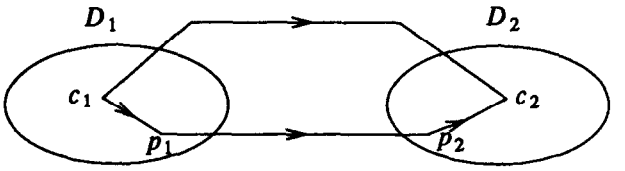

Figure 2.1. Lower bound on distance between two points

$$c_1c_2 \leq c_1p_1 + p_1p_2 + p_2c_2$$

or

$$p_1p_2 \geq c_1c_2 - c_1p_1 - p_2c_2$$

Note that all the three terms on the right hand side of the above inequality have been precomputed for any two points. Therefore, with the data organization described above, the lower bound on distance between any two points in the graph can be easily determined. Also note that this lower bound would be greater than the trivial lower bound of 0 if the two points under consideration are far apart.

2.3 Pruning the Search

Lemma 2.1 can now be used to prune the searches. Let D_1 and D_2 be two distinct domains with centers c_1 and c_2, and we are interested in finding the shortest distance from $p_1 \in D_1$ to $p_2 \in D_2$. An initial upper bound on the shortest distance from p_1 to p_2 can be written as

$$p_1p_2^U = p_1c_1 + c_1c_2 + c_2p_2 \qquad (2.1$$

where p_1c_1 is the shortest distance from p_1 to c_1, c_1c_2 is the shortest distance from c_1 to c_2, and c_2p_2 is the shortest distance from c_2 to p_2. Note that all the terms on the right hand side of the Eqn. (2.1) have been precomputed, and hence this upper bound can be easily determined.

Now suppose that, during the search process, we want to determine whether to open a point p_3 that belongs to a domain D_3, distinct from D_2, and whose center is c_3 (see Figure 2.2). The distance p_1p_3 from p_1 to p_3 would be known at this stage. The point p_3 should be opened only if the distance from p_1 to p_3, p_1p_3, together with the lower bound on the distance from p_3 to p_2, $p_3p_2^L$, is less than the current upper bound on distance from p_1 to p_2, $p_1p_2^U$. That is, only if[3]

2. We have chosen to represent the distance between the points p_1 and p_2 by concatenating the two points, instead of using one symbol with subscripts as in $d_{p_1p_2}$, for notational simplicity.

3. Most search procedures use the best known current distance from p_1 to p_2 as the upper bound on distance from p_1 to p_2. The upper bound on p_1p_2 specified by Eqn. (2.2) is considerably tighter if $p_1p_2^L \neq 0$.

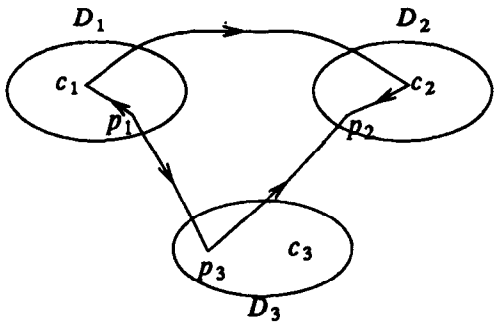

Figure 2.2. Upper bound on distance for opening a point

$$p_1p_3 + p_3p_2^L < p_1p_2^U \quad (2.2$$

By Lemma 2.1

$$p_3p_2^L = c_3c_2 - c_3p_3 - p_2c_2 \quad (2.3$$

Substituting Eqns. (2.1) and (2.3) in (2.2), we obtain the condition as

$$p_1p_3 < (p_1c_1 + c_1c_2 + c_2p_2) - (c_3c_2 - c_3p_3 - p_2c_2)$$

or

$$p_1p_3 < (c_1c_2 - c_3c_2) + p_1c_1 + p_2c_2 + c_2p_2 + c_3p_3 \quad (2.4$$

Our search procedure for determining the shortest distance from p_1 to p_3 can thus be summarized as follows. Start from node p_1 and the upper bound on distance from p_1 to p_2, $p_1p_2^U$, obtained from Eqn. (2.1). Open a point p_3 only if the upper bound on the distance from p_1 to p_3 specified by Eqn. (2.4) is satisfied. If p_3 is opened, we obtain $(p_1p_3 + p_3c_3 + c_3c_2 + c_2p_2)$ as a new bound on distance from p_1 to p_2. If this new bound is lower than the current upper bound on the distance from p_1 to p_2, this bound becomes the new tighter upper bound. Any number of heuristics, such as breadth first, best first, etc. [15], may be used for determining the next candidate point p_3. In the case of Dijkstra's algorithm, the next candidate point is the one that currently has the shortest distance from the starting point (a form of "best first"). Search terminates when no new p_3 may be opened, or the only remaining candidate p_3 is the desired destination p_2 itself.

2.4 Size and Effort Analysis

In this section, we present an approximate analysis to develop an intuitive understanding for the storage overhead due to the precomputed information and savings in effort due to our bounding procedure. The results of the analysis will be confirmed with simulations in Section 4. Storage is measured in units of tuples. A constant multiplication factor, which does not affect the order of magnitude analysis, can be used to convert the measure to bytes or pages. The effort, for the purposes of the analysis, is measured in terms of the number of nodes opened. Once again a multiplication by the average degree will translate it into the number of tuples examined, and does not affect the order of magnitude analysis.

First consider the extra storage required in our scheme. We require extra storage for maintaining the transitive closure of the domain centers, and also for storing the shortest distance between the domain center and all other nodes within a domain. Let us assume that the nodes have been divided into d domains. Thus, there would be d domain centers and their transitive closure would require $O(d^2)$ storage. Since there is an arc between each node and its domain center and vice versa and domains are mutually disjoint, the arcs between domain centers and other nodes in the domain require $O(n)$ storage.

Thus, for a given graph, the data organization that we have presented has an $O(n)$ space overhead and an additional $O(d^2)$ space overhead that depends on the sizes of the domains. By choosing domains to be sufficiently large and hence reducing the number of domains d, the $O(d^2)$ term can be made arbitrarily small and the space overhead can be made within a constant fraction of the storage required for the original relation. However, as we will see shortly, increasing the domain size adversely affects the savings in effort that results from using our bounding procedure.

Turning to the effort analysis, let us define the radius of a domain to be the longest distance between a point in the domain and its center, and let the radius of the domain with largest radius be δ. Then, in the worst case, we can substitute in Eqn. (2.4),

$$p_1c_1 = p_2c_2 = c_2p_2 = c_3p_3 = \delta ,$$

to obtain

$$p_1p_3 + c_3c_2 < c_1c_2 + 4\,\delta \quad (2.5$$

If p_1p_3 is considered approximately equal to c_1c_3, Eqn. (2.5) can be represented as an ellipse with focii c_1 and c_2 and parameters (see Figure 2.3)

$$w = c_1c_2/2, \qquad u = w + 2\,\delta$$

so that

$$v^2 = u^2 - w^2 = 4\,\delta\,(w + \delta) \quad (2.6$$

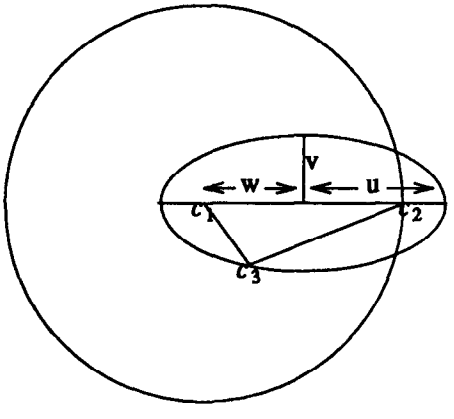

Figure 2.3. Search space with and without domain encoding

Only points lying inside this ellipse are candidates for p_3, with points lying outside not satisfying Eqn. (2.5). Assuming that the points are approximately evenly distributed, the area of this ellipse is the measure of the search space (the number of points that may be explored by the search algorithm), and is given by

$$\pi uv = \pi (w + 2\delta) (4\delta (w + \delta))^{1/4} \simeq 2\pi w (\delta w)^{1/4} \quad (2.7$$

assuming $w \gg \delta$. The implication of this assumption is that we are interested in finding shortest distance between points that are far apart. Thus, the effort using our bounding procedure is $O(w^{3/2} \delta^{1/4})$. This effort increases as the size of the domains is increased, but only as the square root of the radius of the largest domain. Note that this is a worst case analysis, and a domain choice that has a high δ but low average distance from node to domain center may actually perform better for most source-destination choices than one with a somewhat lower δ but most node to domain center distance close to δ.

By way of comparison, observe that the Dijkstra's algorithm proceeds in an expanding circle around the source until it finds the destination. All points inside the circle shown in Fig. 2.3 would be examined by the Dijkstra's algorithm, whereas Dijkstra augmented by our pruning scheme would only consider nodes that lie within the intersection of the circle and the ellipse. Thus, the number of points explored by plain Dijkstra can be approximated by

$$\pi (c_1 c_2)^2 = 4 \pi w^2 \quad (2.8$$

From Eqns. 2.7 and 2.8, the ratio of the number of points explored with and without domain encoding is given by

$$\textit{effort ratio} = (\delta / 4w)^{1/4} = (\delta / 2c_1 c_2)^{1/4} \simeq (\delta / 2p_1 p_2)^{1/4} \quad (2.9$$

If $\delta \ll 2 p_1 p_2$, that is, if we are finding shortest distance between points that are farther apart, the effort ratio will be considerably less than 1 and there would be substantial speed up.

Thus, the ratio of effort in finding shortest distance between two points p_1 and p_2, using our procedure compared to Dijkstra, is $O(\delta^{1/4} p_1 p_2^{-1/4})$. This ratio increases (and hence the speed up reduces) as the square root of the radius of the largest domain, and hence our earlier observation that the benefit of our scheme decreases as the domains are made bigger by decreasing the total number of domains. Notice, however, that while the storage overhead increases as the square of the total number of domains, the effort ratio increases only as the square root of the radius of domains. In Section 4, we will further explore this speed up and the size penalty trade-off as the domain sizes are varied, when we report on the results of experimental evaluation of our bounding procedure.

2.5 Domain Transitive Closures — A Possible Embellishment

Whereas it is not feasible to maintain the entire transitive closure of a large graph, it may be possible to precompute and store the transitive closures of individual domains, particularly if the domains are small. In a sense, that is what we have done at the top level of the data organization presented in Section 2.1 by precomputing the transitive closure of the domain centers (rather than specifying a single center node for the entire graph and maintaining distances between it and the domain centers). The natural question that arises is whether there is any advantage in embellishing the data organization described in Section 2.1 by maintaining transitive closures in the lower level domains as well. We will pursue this embellishment in this section. Note that if the entire domain closure has been computed, distances between points in a domain and their domain center required in the structure described in Section 2.1 are automatically included in the closure, and do not have to be separately stored. However, we will still require the top-level closure of paths between domain centers.

First consider the extra storage requirement of this new data organization. If there are d domains and each domain consists of no more than q points, each domain transitive closure is $O(q^2)$, and the storage required for the domain transitive closures would be $O(d\ q^2)$. In addition to the domain transitive closures, $O(d^2)$ storage would be required for the top level transitive closure. Note that we no longer require distances from the domain center to every other node within a domain. Thus, the total extra storage required is $O(d\ q^2) + O(d^2)$.

Since the number of points in a domain q is $O(n/d)$, the expression for the total extra storage required can be written as $O(n^2/d) + O(d^2)$. This expression is minimized when d is $O(n^{2/3})$, giving a total extra storage requirement of $O(n^{4/3})$. This size is considerably smaller than the size required for storing the closure of entire graph, which is $O(n^2)$. However, $O(n^{4/3})$ could be considerably larger than $O(E)$ storage required for the original graph (E is the number of edges in the graph).

Considering the effort estimates, unfortunately, domain transitive closures do not improve any bounds on the worst-case performance. One can only subjectively state that maintaining local transitive closures may reduce the number of points that need be opened within a domain and thus benefit average performance. Experimental work is required to determine whether the effort savings is substantial enough to offset the extra storage penalty. We will report our experimental results on this count in Section 4.

2.6 Domain Creation

In this section, we address the issue of how to create domains. In many practical situations, there may be information available that automatically suggests how the domains should be structured. For example, on a road map, one would expect to have major intersections and freeway exits as centers of domains that surround them for a certain radius. However, given an arbitrary graph, it is not immediately clear how to form domains. One can think of properties that may be subjectively considered desirable, such as the domains should be roughly equal in radius, should have roughly the same number of nodes, and so on. Let us, therefore, first establish the objective function of interest.

We care about how the domains are formed because they determine how good the bounding procedure is. In particular we obtain an upper bound between two nodes as the sum of the distance between the source and its domain center, the distance between the domain center of the destination and the destination, and the distance between the centers of two domains (Eqn. 2.1). We would like this quantity to be as small as possible (see Eqn. 2.2). On the other hand, we obtain the lower bound between two nodes as the distance between

the centers of the two nodes minus the distance to the source from its domain center minus the distance from the destination to its domain center (Lemma 2.1). We would like this quantity to be as large as possible (see Eqn. 2.2). Thus, there is a conflicting requirement in the case of domain center to domain center distances. On the average, this distance may be a second order effect and can be ignored. However, the distance between a node and its domain center must always be minimized, and reducing this distance on average would improve both bounds.

We can then formally state our problem as one of choosing a specified number of domain centers such that the average distance to (from) a node from (to) its domain center, weighted by the probability of the node occurring in the branch and bound process, is minimized.

For simplicity, let us assume that each node is equally likely to be picked and that the connection to its domain center is equally likely to be required in either direction. If so, our task is to minimize the average distance between the nodes and their domain centers, with the distance in both directions being considered if different. In other words, we wish to select k domain center nodes in a graph with n nodes such as to minimize the sum over all n nodes of the distance from the node to the nearest domain center node. This problem can be shown to be *np*-hard, since a special case of it, for an undirected graph with unit distances on all arcs, and the desired minimum distance being $n-k$, is the well-known *np*-complete vertex-cover problem[4].

We, therefore, developed several heuristics to solve this problem, three of which are described below:

Heuristic I

1. Pick a node at random, not yet member of any domain, and assign it to be the center of a new domain.
2. Assign every node within an empirically selected distance from this domain center[5], not already part of another domain, to belong to the current domain.
3. Repeat steps 1 and 2 until the requisite number of domains have been created.
4. For each node not part of any domain at this point, assign it to the domain whose center is nearest to this node.

For graphs with asymmetric distances between nodes, we average the distance in the two directions.

We tried several variations on this heuristic, none of which improved the performance (in terms of the average distance to center measure), and some of which actually worsened it. Most of these variations were in the nature of rendering the choice of domain centers less random. For example, one could insist upon a certain minimum distance between two nodes selected to be domain centers. Or, for example, one could pick only those nodes to be domain centers that have a low average distance to other nodes.

Heuristic II

1. Pick a node at random, not yet member of any domain, and assign it to be the center of a new domain.
2. Start from the node closest to this domain center and successively consider nodes farther and farther away (in terms of their distance in the transitive closure) until the domain has included its fair share of nodes (empirically varied from the quotient of the total number of nodes in the graph divided by the number of domains specified), or until a node considered is farther away from the domain center than some empirically specified maximum limit. Nodes already assigned to another domain are passed over.
3. Repeat steps 1 and 2 until the requisite number of domains have been created.
4. For each node not part of any domain at this point, assign it to the domain whose center is nearest to this node.

Heuristic III

Both Heuristics I and II require that a complete transitive closure, or at least a large fraction of the closure consisting of the nodes nearest to each node, has been computed prior to domain creation. The following simpler heuristic permits nodes to be assigned to domains without any consideration of the distance labels on the arcs:

1. Pick a node at random, not yet member of any domain, and assign it to be the center of a new domain.
2. Start from nodes that have direct edges (from and) to this domain center and assign them to the current domain if not already assigned. Then consider nodes that are two traversals away, that is, nodes that can reach the domain center through no more than two edge traversals. Then consider the nodes that are three traversals away, and so forth. Between nodes all of which are a certain number of traversals away, consider them in random order. Continue assigning nodes to the current domain one by one until the domain has included its fair share of nodes.
3. Repeat steps 1 and 2 until the requisite number of domains have been created.
4. For each node not part of any domain at this point, consider its immediate neighbors in arbitrary order, then its neighbors two traversals away and so forth, until a node is found that has been assigned a domain. Assign the current node to the same domain.

3. A MULTI-LEVEL STRUCTURE

The domain encoding considered in the previous section can be thought of as dividing the nodes into two "levels". At the

4. We thank Yehuda Afek for showing that this problem is *np*-hard.
5. This empirical distance may initially be set quite large so that every node is assigned a domain even before the requisite number of domains are created. The distance is then gradually reduced until there remained a few nodes that are unassigned after the requisite number of domains are created.

base level (or level 0) there are all the nodes is the graph. At the next level (level 1), sets of these nodes have been aggregated into domains and there is one center node for each domain. The next logical question is whether this idea can be extended to have multiple levels of domains and whether there is any advantage in having a multi-level structure. We will first show how the two-level structure can be extended to multiple levels, and then analyze the effectiveness of such a structure.

3.1 Data Organization

As in the case of two-level structure, divide the data points into domains and identify a center for each domain. However, instead of computing and storing the closure of all paths between centers, divide these centers also into level 2 domains and identify a center for each such domain. These centers are again divided into domains, and so on. At the top level, we have one domain, and we compute and store the closure of all paths between points in that domain. As before, we also maintain distance from a node to its domain center, and vice versa, for domains at all levels. Figure 3.1 pictorially shows the data organization.

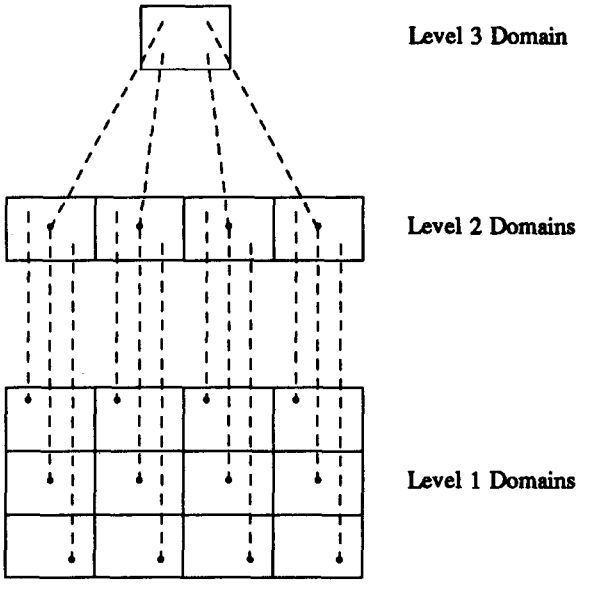

Figure 3.1. Multi-Level Structure. (Boxes represent domains and bullets represent centers)

3.2 Search Procedure

We will now present the bounding procedure that uses the multi-level structure described above to obtain bounds on the point which is being considered to be opened. This bounding procedure can then be incorporated in a branch and bound search algorithm.

We first derive an analog of Lemma 2.1 for the multilevel case that gives a lower bound on the distance between two points that belong to two different domains.

Lemma 3.1. *Let p_1 and p_2 be the two points of interest, and let there be a total of k levels. Let the centers for the domains corresponding to p_1 be c_1^1 (for the level 1 domain), c_1^2 (for the level 2 domain), $\cdots$, c_1^k, and the centers for the domains corresponding to p_2 be $c_2^1, c_2^2, \cdots, c_2^k$. Then,*

$$p_1p_2 \geq c_1^k c_2^k - (c_1^k c_1^{k-1} + c_1^{k-1} c_1^{k-2} + \cdots + c_1^2 c_1^1 + c_1^1 p_1) - (p_2 c_2^1 + c_2^1 c_2^2 + \cdots + c_2^{k-2} c_2^{k-1} + c_2^{k-1} c_2^k)$$

PROOF. Similar to the proof for Lemma 2.1.

Observe that it is possible for the points p_1 and p_2 have a common domain center at level i ($i < k$), and in that case the only realizable lower bound on p_1p_2 would be zero.

The search procedure is similar to the search procedure for two-level structure. An initial upper bound on the distance from p_1 to p_2 is given by

$$p_1p_2^U = (p_1c_1^1 + c_1^1c_1^2 + \cdots + c_1^{k-2}c_1^{k-1} + c_1^{k-1}c_1^k) + c_1^k c_2^k + (c_2^k c_2^{k-1} + c_2^{k-1}c_2^{k-2} + \cdots + c_2^2 c_2^1 + c_2^1 p_2)$$

If p_1 and p_2 have a common center at level i, then the initial upper bound would be

$$p_1p_2^U = (p_1c_1^1 + c_1^1c_1^2 + \cdots + c_1^{i-2}c_1^{i-1} + c_1^{i-1}c_1^i) + (c_2^i c_2^{i-1} + c_2^{i-1}c_2^{i-2} + \cdots + c_2^2 c_2^1 + c_2^1 p_2)$$

A point $p_3 \in D_3$ should be opened only if

$$p_1p_3 < p_1p_2^U - p_3p_2^L$$

The lower bound on distance from p_3 to p_2, $p_3p_2^L$, can be determined using Lemma 3.1.

If a point p_3 is opened, it may result in tightening the upper bound on distance from p_1 to p_3, and the new upper bound may become

$$p_1p_2^U = p_1p_3 + (p_3c_3^1 + c_3^1c_3^2 + \cdots + c_3^{k-2}c_3^{k-1} + c_3^{k-1}c_3^k) + c_3^k c_2^k + (c_2^k c_2^{k-1} + c_2^{k-1}c_2^{k-2} + \cdots + c_2^2 c_3^1 + c_2^1 p_2)$$

if this new bound is lower than the current upper bound. If p_3 and p_2 have a common center at j, then the potential new upper bound would be

$$p_1p_2^U = p_1p_3 + (p_3c_3^1 + c_3^1c_3^2 + \cdots + c_3^{j-2}c_3^{j-1} + c_3^{j-1}c_3^j) + (c_2^j c_2^{j-1} + c_2^{j-1}c_2^{j-2} + \cdots + c_2^2 c_3^1 + c_2^1 p_2)$$

3.3 Size and Effort Analysis

We will first informally and then formally argue that the multi-level structure just presented is not a viable alternative to the two-level structure presented in Section 2. The problem with the multi-level structure is that it considerably weakens the bounding procedure. If the source and destination points are nearby, then the lower bounds generated by Lemma 3.1 would almost always be zero. On the other hand, if the source and destination points are far apart, then the upper bounds become very loose as the effective radius increases. There is some reduction in storage overhead since the number of points at the top level would be smaller than the number of points in the two-level structure, and hence the top level closure would be smaller. However, we will now have to additionally keep distances from points in the intermediate levels to their domain centers at the higher levels.

To see this formally, let us treat the entire graph as a single domain at level k. Let there be d_{k-1} domains at level $k-1$, each of which contains d_{k-2} domains at level $k-2$, giving a total of $d_{k-1}d_{k-2}$ domains at level $k-2$. Similarly, let there be $d_{k-1}d_{k-2}\cdots d_{k-j} = \prod_{i=1}^{j} d_{k-i}$ domains at level $k-j$. For

simplicity of notation, consider each node of the original graph to be in a level 0 domain by itself, with the node itself being the domain center. Now since level 0 domains are the individual nodes themselves, the total number of nodes in the graph $n = d_{k-1}d_{k-2}\cdots d_{k-k} = \prod_{i=1}^{k} d_{k-i} = \prod_{i=0}^{k-1} d_i$. Note that in the case of two-level structure, we had only a d_0, which we called q, and a d_1 we called d.

Let us first compute the extra storage requirement. For simplicity, let $d_0 = d_1 = \cdots = d_{k-1} = d$. We maintain closure only for the level $k-1$ domain centers, and need $O(d^2)$ storage for it, since there are d level $k-1$ domain centers. In addition, for each domain, we keep distance from the points in the domain to the domain center and vice versa. Provided $d \gg 1$, we need only consider this storage at the lowest level. At all higher levels, there are significantly fewer nodes. So we need $O(n)$ storage for distance between nodes and domain centers. Thus, the total extra storage required is $O(n) + O(d^2)$. This expression is similar to the two-level case, the only difference being d is likely to be significantly smaller now, since $d = n^{1/k}$ for the multi-level case whereas $d = n^{1/2}$ for the two-level structure. The storage goes down as k increases, but only very slowly.

Let us now examine the effect on effort. Let δ_1 through δ_{k-1} be the maximum radii of the level 1 through level $k-1$ domains, and let $\Delta_j = \sum_{i=1}^{j} \delta_i$. Then, the effective maximum "radius" of a level $k-1$ domain, which is the upper bound on the path we generate through our storage structure from a node to its $k-1$ level domain center, is Δ_{k-1}. In the worst case, the nodes between which bounds are sought, belong to different domains except at the top level. Therefore, the effort computation can be made in the same fashion as for a two-level data organization, with Δ_{k-1} used as the radius. Thus, the effort required is bounded solely by the radius of the top-level domains. Since, Δ_{k-1} is large, we get very poor bounding with the multi-level structure.

We indeed performed several simulations (experimental results not reported in this paper) and the multi-level structure just described was found to consistently perform worse than the two-level structure. As such, this form of multi-level encoding will not be discussed any further.

3.4 An Embellishment

We saw that the multi-level structure has better storage characteristics than the two-level structure, but has poor bounding characteristics. We will now describe an embellishment of the above multi-level structure that incurs slightly higher storage penalty but has the potential of exhibiting better bounding characteristics. The basic idea is to keep the domain closures at every level instead of only at the top level, as was done in embellishing the two-level structure in Section 2.

With this embellishment, the lower bound on distance between two points p_1 and p_2 is given by

$$p_1p_2 \geq c_1^i c_2^i - (c_1^i c_1^{i-1} + c_1^{i-1} c_1^{i-2} + \cdots + c_1^2 c_1^1 + c_1^1 p_1) - (p_2 c_2^1 + c_2^1 c_2^2 + \cdots + c_2^{i-2} c_2^{i-1} + c_2^{i-1} c_2^i)$$

where i ($i \leq k$) is the level at which there exists a domain such that the shortest distance between c_1^i and c_2^i has been stored.

The initial upper bound on the distance from points p_1 to p_2 is given by

$$p_1p_2{}^U = (p_1 c_1^1 + c_1^1 c_1^2 + \cdots + c_1^{j-2} c_1^{j-1} + c_1^{j-1} c_1^j) + c_1^j c_2^j + (c_2^j c_2^{j-1} + c_2^{j-1} c_2^{j-2} + \cdots + c_2^2 c_2^1 + c_2^1 p_2)$$

where j is the smallest level at which the shortest distance between c_1^j and c_2^j has been stored for some j.

Finally, if a point p_3 is opened, it may potentially tighten the upper bound on distance from p_1 to p_3, and the new upper bound may become

$$p_1p_2{}^U = p_1p_3 + (p_3 c_3^1 + c_3^1 c_3^2 + \cdots + c_3^{l-2} c_3^{l-1} + c_3^{l-1} c_3^l) + c_3^l c_2^l + (c_2^l c_2^{l-1} + c_2^{l-1} c_2^{l-2} + \cdots + c_2^2 c_3^1 + c_2^1 p_2)$$

where l is the smallest level at which the shortest distance between c_3^l and c_2^l has been stored for some l.

Let us now analyze the effect of this embellishment.

First note that each domain has d points, and hence the size of transitive closure local to each domain is $O(d^2)$. The total number of domains is dominated by the number of domains at the lowest level, which is $O(n/d)$. Therefore, the additional storage required for domain closures is $O(nd)$. We do not require distances from center to nodes within domain, as these distances are already included in the closure of the domains. By choosing k to be sufficiently large, d and hence the factor nd can be made arbitrarily small. Thus, the additional storage overhead can be reduced to no more than a constant factor of the storage required for the original relation.

The domain closures do not reduce the radius of the top level domain, and hence the worst case effort continues to be as bad as for the multi-level structure without domain closures. However, it is now possible that whenever the nodes in question lie within the same domain well below the top level, much tighter bounds may be obtained. Only experimental study can tell whether this trade-off is reasonable. We will present these experimental results in Section 4.

Before leaving the topic of the multi-level structures, we would like to make a few points in their favor. The reason the multi-level structure did so badly in our analysis is that for large k, the top level domains are very large and hence provide very weak bounds. The effectiveness of multi-level structures would be enhanced if the size of the $k-1$ level domains is kept considerably smaller than the size of the graph. These $k-1$ size domains could then be split into lower level domains that may or may not be considerably smaller. Moreover, in a situation in which most queries concern points that are not far apart, a multi-level structure may be a good choice. Finally, an attractive use of multi-level structure is in obtaining approximate solutions with great savings in effort as discussed in [2].

4. PERFORMANCE EVALUATION

In this section, we present the results of several simulation experiments that we performed to study the effectiveness of the data organization techniques presented in this paper. We

first make a few observations on the performance evaluation methodology, and describe the datasets used in the study.

4.1 Methodology

We use two performance metrics. One is the *size ratio* which is defined to be the ratio of the size of total information stored with our domain encoding technique and the size of original database. One would like this metric to be as close to 1 as possible. The other metric is the *effort ratio* which is defined to be the ratio of the I/O by the Dijkstra algorithm with and without domain encoding. The effort with domain encoding includes the I/O for fetching the bounding information. The effort has been computed by considering search for shortest distance between several points with varying distance between them, and averaging over all searches. Care was taken to ensure that domain centers were not chosen to be the source or destination for any search. If a domain center were the source or destination, our encoding structure would result in considerably tighter bounding and consequently in much less effort. The Dijkstra algorithm has been used as the benchmark, since it is generally regarded to be the best algorithm for finding shortest path between two points [11]. One would like to make the effort ratio as close to zero as possible.

Following the lead in [4], synthetic graphs were used as data sets. The distance between two nodes was assumed to be a uniform random variable over a specified positive interval. The number of nodes were varied to obtain databases of different sizes, and for a given database, the number of domains were varied to get domains of different sizes. Heuristic III given in Section 2.6 was used to divide the nodes into various domains. We chose Heuristic III for its computational simplicity. The other two heuristics should result in even better performance of our scheme. Most of the experiments were performed with a graph of 2500 nodes, with average outdegree of 8 and the average distance value of 5. We couldn't use larger databases in the simulations as that would have made simulations prohibitively expensive to run. However, our analysis indicates that the larger the database the more effective our techniques should be.

4.2 Experiment 1: Two-Level Structure without Domain Closures

In the first set of experiments, the effort and size ratios were measured as a function of domain size for the two-level structure without domain closures. The domain sizes have been specified in number of nodes in a domain. Figure 4.1 shows the result for the 2500 node database.

For very small domain sizes, there is significant size overhead. Small domain sizes result in a large number of domains, and hence the storage required to maintain distance between every pair of domain centers become large. As the domain size is increased, the size overhead decreases, and for large domains, the size overhead becomes a constant fraction of the size of the original relation. The effort ratio, on the other hand, increases as the square root of the domain size as the domain size is increased. One could then choose an operating point that gives a large speed-up at the expense of large storage overhead, or alternately, if the storage is at

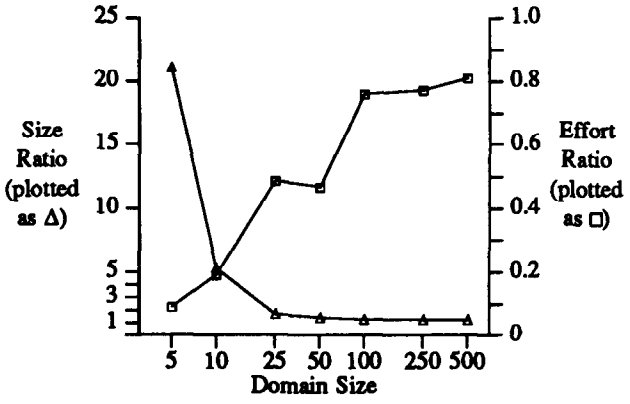

Figure 4.1. Effort and Size ratios for two-level structure (2500 node database)

premium, one could pay a small storage overhead and still get some speed up.

A good choice for the domain size, which achieves good speed up and incurs only moderate storage overhead, seems to be the square root of the total number of nodes in the graph. Figure 4.2 shows, for this choice of domain size, the effort and size ratios for graphs of different sizes. Database size has been expressed in number of nodes in the corresponding graph.

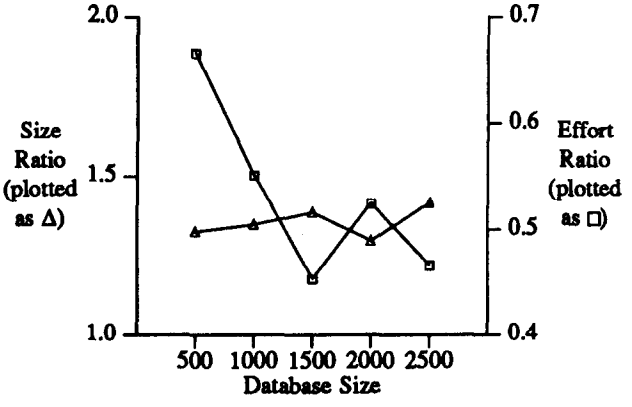

Figure 4.2. Effort and Size ratios for two-level structure (different databases)

This graph is very encouraging as it shows that by paying about 40% storage overhead, nearly 100% speed up may be obtained.

4.3 Experiment 2: Two-Level Structure with Domain Closures

The second set of experiments examined the usefulness of precomputing the transitive closure within each domain as suggested in Section 2.5. The effort and size ratios have been plotted in Figure 4.3 as a function of domain size for the same 2500 node database.

When the domain sizes are small, very little storage is required to store the domain closures. However, there are a large number of domain centers, and the total storage requirement is dominated by the closure between these domain centers. For large domain sizes, there are very few domain

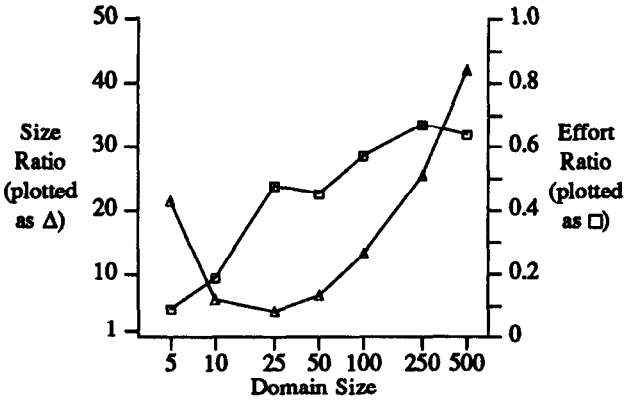

Figure 4.3. Effort and Size ratios for two-level structure with domain closures (2500 node database)

centers, and the closure between them would be small. However, the domain closures now become very large. As discussed in Section 2.5, the optimum domain size is of the order of the cube root of the number of nodes. The shape of the effort ratio curve is the same as in Figure 4.1 for the two-level structure without domain closures. This is not very surprising, given our observation in Section 2.5 that no significant improvement in bounding is obtained by keeping the domain closures.

Figure 4.4 presents a comparison of the two-level structure with and without domain closure. In this figure, we have plotted the effort ratio against the size ratio for two schemes. The various data points in this figure have been obtained by extracting effort ratio and size ratio numbers for different domain values from Figures 4.1 and 4.3. Note from Figure 4.3 that two different effort ratios are obtained, one higher than the other, for the same size ratio, due to the concavity of the size ratio curve. This accounts for the rather odd shape of the curve for the structure with domain closures in Figure 4.4.

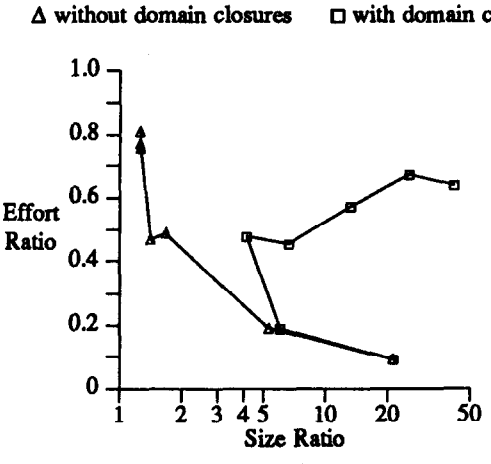

Figure 4.4. Effort vs. Size for two-level structure with and without domain closures (2500 node database)

It is apparent from Figure 4.4 that performance-wise the scheme without domain closures totally dominates the scheme with domain closures. For any acceptable storage overhead, better speed up may be obtained using the structure without domain closures. Similarly, for any desired speed up, the without domain closures structure incurs less storage overhead. Although not presented here, similar results were also obtained with databases of different sizes. We can thus conclude that the effort saving resulting from a reduction in number of points that need be opened in a domain does not justify the large storage overhead incurred by domain closures.

4.4 Experiment 3: Multi-Level Structure

The third set of experiments examined the effectiveness of the multi-level structure with domain closures presented in Section 3. The experiments were performed for the 2500 node database and 4 nodes per domain (except the top level), and by varying the number of levels in the structure. The small domain sizes were chosen to obtain sufficient number of levels. Figure 4.5 shows the effort and size ratios for different number of levels in the multi-level structure.

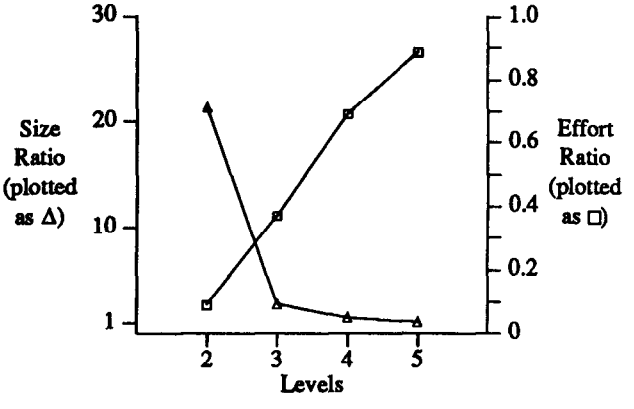

Figure 4.5. Effort and Size ratios for multi-level structure with domain closures (2500 node database)

For two levels, the multi-level structure reduces to a two-level structure with domain closures such as the one discussed in the previous experiment. This data point is plotted simply to provide a reference. For other levels, as predicted in Section 3, the size overhead is considerably reduced, but at the same time, the effort ratio also increases.

By way of comparison, we have plotted the effort vs. size curves for the two-level structure and the multi-level structure in Figure 4.6 It can be seen that performance-wise, for most of the operating region, the two-level structure dominates the multi-level structure. However, it is possible to reduce the storage overhead at a level which is not possible with the two-level structure at considerable loss in speed ups. Note that the two-level structure requires at least two times the number of nodes units of additional storage (to store distance from each node to its domain center and vice versa), whereas with the multi-level structure one could go below this bound on storage overhead.

In order to ensure that the trends that we got for multi-level structure have not been biased by our choice of domain size, we obtained the effort and size ratios for three-level structure for different domain sizes and have compared it with the two-level structure in Figure 4.7.

It is clear from Figure 4.7 that the two-level structure completely dominates the three-level structure. The odd shape

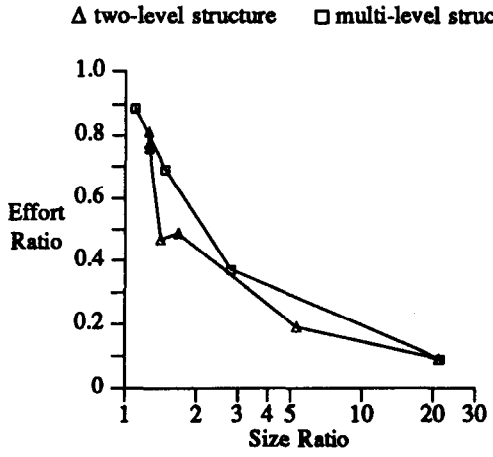

Figure 4.6. Effort vs. Size for two-level structure without domain closures and multi-level structure with domain closures (50 × 50 database)

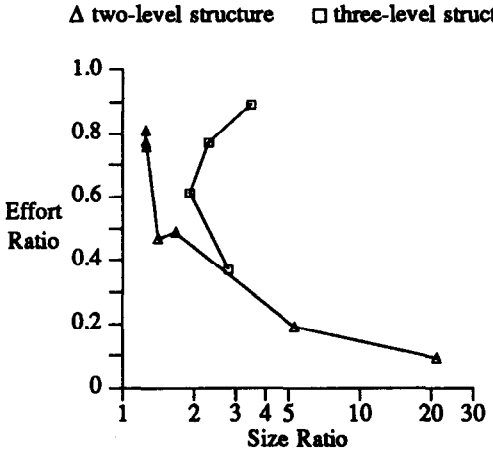

Figure 4.7. Effort vs. Size for two-level structure without domain closures and three-level structure with domain closures (2500 node database)

of the curve for the three-level structure in Figure 4.7 is due to the concavity of the size ratio as the domain size is varied. The storage required for the top level closure dominates the storage overhead for small sized domains. For large domain sizes, the closures at lower levels dominate the storage overhead.

We can thus conclude that the justification for the multi-level structure stems not from speed up consideration but the storage overhead consideration. Only if the storage is at a premium does a multi-level structure become attractive, with very small domains and a large number of levels.

4.5 Summary of Experimental Results

From the simulation results presented in this section, the two-level structure without domain closures emerges as the data organization technique of choice. It offers a wide range of operating points to choose from depending upon the speed up desired and the storage overhead one is willing to incur. A good choice for the domain size, which achieves significant speed up and incurs only moderate storage overhead, seems to be the square root of the database size. For this choice, we were able to obtain nearly 100% reduction in I/O by paying about 40% disk storage overhead. Note that the effort calculation with our domain encoding scheme included extra I/O to fetch the necessary bounding information. Considering the fact that the large databases are generally I/O bound, the significant reduction in I/O due to the search space pruning makes our scheme very attractive.

There doesn't seem to be any advantage in keeping the domain closures with the two-level structure. The small additional savings in I/O resulting from a reduction in number of points that need be opened in a domain does not justify the large storage overhead incurred by domain closures.

If one is primarily interested in speed up, the multi-level structure also is not a viable alternative. However, if the storage is at a premium and one is interested in obtaining some speed up by paying very little storage overhead, a multi-level structure may be used. Note that the two-level structure requires at least two times the number of nodes units of additional storage (to store distance from each node to its domain center and vice versa), whereas with the multi-level structure one could go below this bound on storage overhead.

5. GENERALIZATIONS

In this section, we present some generalizations of the techniques presented in the previous sections. In particular, we show in Section 5.1 how our techniques apply to problems other than shortest path problems and search algorithms other than Dijkstra's algorithm. In Section 5.2, we consider the case when the domains are not mutually disjoint. Finally, in Section 5.3, we suggest how to handle gracefully changes to the base relation that could invalidate precomputed shortest paths stored as part of our data structure.

5.1 Other Applications

So far in this paper, all the discussion has been centered around the problem of determining the shortest path between two points and how Dijkstra's algorithm can be speeded up using our bounding procedure. However, as stated in Section 1, the techniques presented in this paper apply equally well to all extremal path problems, and our bounding procedure can be incorporated in any search algorithm based on the state-space search paradigm. In this section, we illustrate how these generalizations are possible.

An extremal path problem on a graph involves the identification of a path between a pair of nodes in the graph that has an extreme value (highest or lowest on some precedence ordering) for its label, or the calculation of the value of such an extremal label. If one is interested in smallest or lowest value, the bounding procedure developed for the shortest path problem directly applies with distance being replaced by the appropriate quantity. For largest or highest value, the bounding procedure can easily be modified by switching the roles of upper and lower bounds. We illustrate using the problem of determining the longest path between two points as the paradigm for deriving the bounding procedure, and then we will incorporate this bounding procedure in a breadth-first search algorithm.

The database will again have to be divided into domains. However, we will now maintain largest distance between domain centers, and between the domain center and all other points and vice versa within a domain. We discuss only the two-level structure.

With this data organization, first of all, an initial lower bound on the largest distance between the points of interest, p_1 and p_2, is obtained as:

$$p_1p_2^L = p_1c_1 + c_1c_2 + c_2p_2$$

where c_i is the domain center of the domain D_i to which belongs the point p_i. During the search process, a point p_3 should be opened only if

$$p_1p_3 + p_3p_2^U > p_1p_2^L \quad (5.1$$

By a reasoning similar to Lemma 2.1, an upper bound on distance between p_3 and p_2 can be obtained as:

$$p_3p_2^U = c_3c_2 - c_3p_3 - p_2c_2$$

We will now incorporate the above bounding procedure in a breadth-first search [15] procedure. Note that the semi-naive algorithm [6] also performs a breadth-first search for determining reachability from a specified node. In the following algorithm, OPEN is a queue, each element of which is a tuple of the form <node, distance> where the distance field contains the best (largest) known distance from source to the corresponding node.

```
/*
 * Breadth-first Search with bounding for determining
 * largest distance between points p and q
 */

determine the initial lower bound on largest distance, pq^L

OPEN := <p, 0>

while q is not the only element in OPEN do
 {
  remove the first element <i, d_i> from OPEN (other than q);
  for every j ∈ Succ(i) do
   {
    if j is in OPEN then
       <j, d_j> := <j, max(d_j, d_i+d_ij)>
    else do
     {
      determine if j should be opened  -- use Eqn. (5.1)
      if j needs to be opened then
       {
        append <j, d_i+d_ij> to OPEN;
        update pq^L
       }
     }
   }
 }
```

5.2 Multiple Domain Membership

We have thus far assumed that the nodes have been divided into non-intersecting domains, so that each node has a unique domain center. If a node is allowed to belong to more than one domains, there will be multiple domain centers that can be reached from a node. For each pair of domain centers selected (one for the source, one for the destination node) a bound is obtained on the path that we wish to bound. Several such pairs are considered and the one that produces the tightest bound is the one that is selected. The advantage is that considerably tighter bounds can be obtained. The disadvantage is that if each node has c domain centers, c^2 bounds have to be considered, and unless c is kept small, the effort involved in simply bounding the search could become significant.

5.3 Incremental Changes

Whenever some derived information is materialized, a change in the base information needs to be reflected in a change in the derived information. We require precomputed shortest distances between domain centers, and between each domain center and its constituent nodes. Whenever a modification is made to the original graph, this precomputed information has to be updated. Obviously, a complete recomputation would be extremely expensive. One possibility is to use the incremental techniques suggested in [1]. However, given the extremely large sizes of graphs that we now have in mind, even these incremental techniques may be too expensive to use frequently. Fortunately, a simple solution exists.

The basic observation to make is that the precomputed shortest distances are needed to derive bounds that are used to prune the search. Even if we did not have exact values of these shortest distances, but rather only upper and lower bounds on them, these bounds can appropriately be used in place of exact values, while deriving bounds for pruning the search. Thus, instead of maintaining precomputed shortest distances between domain centers and between each domain center and its constituent nodes, we will maintain the upper and lower bounds on these distances. To begin with, the upper and lower bounds would be same (and equal to exact distances). As the base relation is updated, instead of recomputing the materialized shortest distances, we will appropriately update the upper or the lower bound. Of course, we will get somewhat less pruning since we now have weaker bounds than we would if we knew the exact distances. After several modifications to the database, the upper and lower bounds on precomputed distances would diverge quite a bit, and the exact shortest distances required by the data structure may be recomputed. In a quasi-static situation, this approach can become very attractive.

6. CONCLUSIONS

In this paper, we considered the problem of performing efficient search over large databases. To this end, we presented a data organization technique that relies on partially precomputing some information, and a bounding procedure that uses this data organization to prune the search space. Our data organization technique and the bounding procedure may be incorporated in a branch and bound search algorithm, or new algorithms can be designed using our bounding procedure. These techniques can be used to solve a large number of useful and practical path problems such as the shortest path, critical path, largest capacity path, path of maximum reliability, etc. [3, 5, 8, 9, 14, 16], and can also be

gainfully employed in large expert database systems in which the search component of the expert systems has been integrated with the data management capability of database management systems. Simulation results confirm that, using these techniques, a search can be expedited significantly without incurring a large storage penalty.

REFERENCES

[1] R. Agrawal and H. V. Jagadish, "An Efficient Method for Encoding Path Information in the Transitive Closure of a Database Relation", Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, Nov. 1987.

[2] R. Agrawal and H. V. Jagadish, "Efficient Search in Expert Database Systems", Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, Nov. 1987.

[3] 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).

[4] 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.

[5] R. Agrawal, S. Dar and H. V. Jagadish, "Transitive Closure Algorithms Revisited: The Case of Path Computations", Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, Jan. 1988.

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

[7] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, William Kaufman, Los Altos, California, 1981.

[8] J. Biskup, U. Raesch and H. Stiefeling, "An Extended Relational Query Language for Knowledgebase Support", Institut fuer Informatik, Hildesheim, West Germany, 1987.

[9] B. Carre, *Graphs and Networks*, Clarendon Press, Oxford, 1978.

[10] E. W. Dijkstra, "A Note on Two Problems in Connection with Graphs", *Numer. Math. 1*, (1959), 269-271.

[11] S. E. Dreyfus, "An Appraisal of Some Shortest Path Algorithms", *Operations Res. 17*, 3 (1969), 395-412.

[12] 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.

[13] 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.

[14] T. H. Merrett, *Relational Information System*, Reston Publishing, Reston, Virginia, 1984.

[15] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison Wesley, Reading, Massachusetts, 1984.

[16] 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.