Algorithms for Searching Massive Graphs
Rakesh Agrawal and H. V. Jagadish

Abstract—Given a large graph, stored on disk, there is often a need to perform a search over this graph. Such a need could arise, for example, in the search component of a data-intensive expert system or to solve path problems in deductive database systems. In this paper, we present a novel data structuring technique and show how a branch and bound search algorithm can use this data structuring 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.

Index Terms—Deductive database systems, search techniques, branch and bound, approximation techniques, path queries, query processing, shortest distance in a graph.

I. INTRODUCTION

We investigate how a search can be performed efficiently over massive graphs that are too big to fit in memory and must be stored on disk. Such a capability is important for deductive databases that are used for solving extremal path problems. Extremal path problems [10] constitute a large and useful subclass of recursive queries and arise in several practical applications [1], [8], [27]. 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 subpaths) 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 major 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 [1], [5], [8], [10]-[12], [24], [27], and [29].

It has been observed [23] that a significant part of a wide variety of expert systems is the component that performs search over a sizable data space. For large data sets, a database management system is a practical necessity, and therefore it is prudent that the search portion of an expert system be moved into the database management system. The major advantage of this approach is that the search functions can be written and optimized once within the database management system, instead of writing them separately for every expert system that is coupled with the database management system.

We discuss issues of both data organization as well as 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 search problems.

Our overall approach is to precompute some partial information and then 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 [23], one can estimate that simply to store a small 100-mi x 100-mi map discretized at 100-ft intervals, one requires about 2.4 Gbytes 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 of this magnitude, precomputing and storing path information, even after using the encoding and compression techniques that are proposed in [2], [3], and [21], 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 opened1 during the search process. These bounds can cut down very significantly the number of data points explored by a search algorithm, such as Dijkstra's algorithm [13] for finding the shortest path between two points.

The issue of integrating the search function of expert systems with the database management system has also been addressed in [23]. However, the two approaches are quite different. The approach taken in [23] was to adopt Guttman's extension to QUEL with a * operator [17] so that various search algorithms may be expressed in this extended language, christened QUEL*. The effect of the * operator was simulated on the relational technology version of INGRES by rerunning a command until no new tuples were generated. Thus, in this approach, the integration of the database management system and the search component is, at best, quite loose. A standard relational database system is used, nothing special is done to organize the data in a way that would speed up the search2, and the search is performed by repeated executions of standard QUEL commands. We, on the other hand, are

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 Of course, fast access structures, such as indexes, could be used.
proposing a much closer integration of the search component and the database management system.\(^3\)

Join indices \([22], [30]\) and their use in computing transitive closure \([31]\) can also be viewed as the partial precomputation of results. However, our proposal is much more aggressive in terms of storage and what is kept precomputed. Join indices may speed up a transitive closure computation by an iterative algorithm, such as the seminaive or logarithmic algorithm \([20], [31]\) that repetitively performs joins in a loop. Our bounding procedure prunes out paths that are otherwise joinable but would not lead to the desired solution, and as such can be incorporated in the iterative algorithms with and without join indices. Indeed, later in the paper, we will give an example of how our bounding procedure can be used to speed up determination of longest path using the seminaive algorithm.

In \([27]\), the need for traversing graphs and computing properties (such as length of shortest path) has been eloquently presented. Our paper here can be viewed as a description of one way in which some of the computations described there could be performed more efficiently.

The organization of the rest of the paper is as follows. In Section II, 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 multilevel structure in Section III. In Section IV, we show how our data organization scheme can be used to generate approximate solutions with considerable reduction in effort. In Section V, 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 VI. We present our conclusions in Section VII.

II. DOMAIN ENCODING

In the state-space search paradigm \([7]\), 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 the 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 (see \([25]\)) to improve its performance.

Our bounding procedure is based on partially precomputing some information and then using it to develop 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.

\(^3\)A fair criticism is that we do it at the expense of some flexibility and extensibility.

A. Data Organization

Given a directed or undirected graph consisting of nodes, arcs between the nodes, and non-negative 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 II-VI. 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 \([2], [3]\), and \([21]\). Efficient techniques, such as those described in \([4]\) and \([5]\), 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)\(^4\) is precomputed and stored.

B. A Lower Bound

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

\textbf{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 \(p_2\) is

\[ p_1 p_2 \geq c_1 c_2 - c_1 p_1 - p_2 c_2, \]

where \(c_1 c_2\) is the shortest distance from \(c_1\) to \(c_2\), \(c_1 p_1\) is the shortest distance from \(c_1\) to \(p_1\), and \(p_2 c_2\) is the shortest distance from \(p_2\) to \(c_2\).\(^5\)

\textbf{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 \(c_1 c_2\) 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 Fig. 1)

\[ c_1 c_2 \leq c_1 p_1 + p_1 p_2 + p_2 c_2 \]

or

\[ p_1 p_2 \geq c_1 c_2 - c_1 p_1 - p_2 c_2. \]

All 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 easily be determined.

\(^4\)In general, there could be different distances between a pair of nodes depending on the direction traversed. This may be the case, for example, when measuring driving distance on a road map with one-way streets, or when measuring driving times on a road map during rush hour when traffic one way may be moving much more slowly than traffic in the other direction.

\(^5\)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 \(d_{p_1 p_2}\) for notational simplicity.
Also note that this lower bound is greater than the trivial lower bound of 0 if the two points under consideration are far apart.

C. Pruning the Search

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

\[
p_{1P_2}^U = p_1c_1 + c_1c_2 + c_2p_2,
\]

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 (1) have been precomputed, and hence this upper bound can easily be 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 Fig. 2). The distance \( p_1p_3 \) from \( p_1 \) to \( p_3 \) is 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 \( p_2 \), \( p_3p_2 \), is less than the current upper bound on distance from \( p_1 \) to \( p_2 \), \( p_{1P_2}^U \). That is, only if it holds

\[
p_1p_3 + p_3p_2^L < p_{1P_2}^U.
\]  

By Lemma 2.1,

\[
p_3p_2^L = c_3c_2 - c_3p_3 - p_2c_2.
\]

Substituting (1) and (3) in (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

\[
p_1p_3 < (c_1c_2 - c_3c_2) + p_1c_1 + p_2c_2 + c_2p_2 + c_3p_3.
\]  

Our search procedure for determining the shortest distance from \( p_1 \) to \( p_2 \) 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 (1). Open a point \( p_3 \) only if the upper bound

\[
6Most search procedures, such as [26], use the best known current distance from \( p_1 \) to \( p_3 \) as the upper bound on distance from \( p_1 \) to \( p_3 \). The upper bound on \( p_1p_3 \) specified by (2) is considerably tighter if \( p_3p_2^L \neq 0 \).

on the distance from \( p_1 \) to \( p_3 \) specified by (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 (tighter) than the current upper bound on the distance from \( p_1 \) to \( p_2 \), this bound becomes the new upper bound. Any heuristic, such as breadth first, best first, etc., may be used for determining the next candidate point \( p_3 \) [26]. 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.

The A* algorithm [19] also addresses the problem of finding a minimal path between two nodes in a state-space graph. It is an ordered state-space search algorithm that uses an evaluation function \( f \) and chooses for expansion a node \( n \) for which \( f \) has the smallest value among all nodes that have not been expanded so far. The function \( f \) is defined as

\[
f(n) = g(n) + h(n),
\]

where \( g \) estimates the minimum cost of a path from the start node to node \( n \), and \( h \) estimates the minimum cost from node \( n \) to the goal node. The bounds presented above can also be embedded in an A* algorithm. While finding the shortest distance from \( p_1 \) to \( p_2 \), we have on hand for any intermediate node \( p_3 \) the estimate \( g(p_3) \) and we can use (3) to estimate \( h(p_3) \).

D. 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 V. 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,
where n is the total number of nodes in the graph.

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 result
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 \( \delta \). Then, in the worst case, we can substitute
in (4),

\[
p_1c_1 = p_2c_2 = c_3p_3 = \delta,
\]

to obtain

\[
p_1p_3 + c_3c_2 < c_1c_2 + 4\delta.
\]

(5)

If \( p_1p_3 \) is considered approximately equal to \( c_1c_3 \), (5) can be
represented as an ellipse with focii \( c_1 \) and \( c_2 \) and parameters
(see Figure 3)

\[
w = c_1c_2/2, u = w + 2\delta
\]
so that

\[
v^2 = u^2 - w^2 = 4\delta(w + \delta)
\]

(6)

Only points lying inside this ellipse are candidates for \( p_3 \),
with points lying outside not satisfying (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

\[
\pi w = \pi(w + 2\delta)(4\delta(w + \delta))^{1/2} \approx 2\pi w(\delta w)^{1/2},
\]

(7)

assuming \( w >> \delta \). The implication of this assumption is
that we are interested in finding the shortest distance between
points that are far apart. Thus, the effort using our bounding
procedure is \( O(w^{3/2}\delta^{1/2}) \). 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 \( \delta \) but low
average distance from node to domain center may actually
perform better for most source-destination choices than one
with a somewhat lower \( \delta \) but most node to domain center
distance close to \( \delta \).

By way of comparison, observe that Dijkstra’s algorithm
proceeds in an expanding circle around the source until it
finds the destination. All points inside the circle shown in
Fig. 3 would be examined by Dijkstra’s algorithm, whereas
Dijkstra’s algorithm augmented by our pruning scheme would
consider only nodes that lie within the intersection of the circle
and the ellipse. Thus, the number of points explored by a plain
application of Dijkstra’s algorithm can be approximated by

\[
\pi(c_1c_2)^2 = 4\pi w^2.
\]

(8)

From (7) and (8), the ratio of the number of points explored
with and without domain encoding is given by

\[
\text{effort ratio} = (\delta/4w)^{1/2} = (\delta/2c_1c_2)^{1/2} \approx (\delta/2p_1p_2)^{1/2},
\]

(9)

If \( \delta \ll 2p_1p_2 \), 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 the shortest distance be-
tween two points \( p_1 \) and \( p_2 \), using our procedure compared to
Dijkstra’s algorithm, is \( O(\delta^{1/2}/p_1p_2)^{1/2} \). 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. Note,
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 V,
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.

E. 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 II-
A 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 II-A 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 II-A 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 \( O(dq^2) \). In addition to the domain transitive closures, \( O(d^2) \) storage would be required for the top-level transitive closure. Thus, the total extra storage required is \( O(dq^2) + O(d^2) \).

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 V.

F. 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 subjectively be considered desirable, such as that 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 the goodness of the bounding procedure. 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 (1). We would like this quantity to be as small as possible (see (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 (2)). Thus, there is a conflicting requirement in the case of domain center to domain center distances. On 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 the sake of 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 (see [16]) problem.

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

**Heuristic 1:**
1) Pick a node at random, not yet a 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, 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.

**Heuristic 2:**
1) Pick a node at random, not yet a member of any domain, and assign it to be the center of a new domain.

---

\(^7\) We thank Yehuda Afek for showing that this problem is np-hard.
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 3:**

1) Pick a node at random, not yet a 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) 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 this node to the same domain.

Observe that Heuristic 3 permits nodes to be assigned to domains without any consideration of the distance labels on the arcs. On the other hand, both Heuristics 1 and 2 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 fact that Heuristic 3 does not require a precomputed transitive closure renders it extremely attractive compared to the other two. In fact, for very large graphs, this may be the only heuristic of the three that is applicable, since the transitive closure is too expensive (perhaps practically impossible) to compute. As such, Heuristic 3 is used for domain creation where required in this paper, particularly in the simulation experiments described in Section V.

III. A MULTILEVEL STRUCTURE

The domain encoding considered in the previous section can be thought of as dividing the nodes into two "levels." All the nodes in the graph are at the base level (or level 0). 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 multilevel 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.

A. 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. Fig. 4 pictorially shows the data organization.

B. Search Procedure

We will now present the bounding procedure that uses the multilevel structure described above to obtain bounds on the point that 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), \( c_2^1 \) (for the level 2 domain), \( \ldots \), \( c_k^1 \), and the centers for the domains corresponding to \( p_2 \) be \( c_1^2 \), \( c_2^2 \), \( \ldots \), \( c_k^2 \). Then,

\[
p_{12} \geq c_k^1 c_k^2 - (c_k c_{k-1}^2 + c_{k-1} c_{k-2}^2 + \cdots + c_2^1 c_1^2 + c_1^1 p_1)
- (p_2 c_k^2 + c_k^2 c_{k-2}^2 + \cdots + c_2^2 c_2^1 + c_2^1 c_2^1).
\]

**Proof:** Similar to the proof for Lemma 2.1. Observe that it is possible for the points \( p_1 \) and \( p_2 \) to have a common domain center at level \( i \) (\( i < k \)), and in that case the only realizable lower bound on \( p_1 p_2 \) 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
\[
p_1p_2^L = (c_1^1 + c_1^2 + \cdots + c_1^{k-2}c_1^{k-1} + c_1^{k-1}c_1^k) + c_1^k + \\
(c_1^{k-1}c_2 + c_2^{k-1}c_2^{k-2} + \cdots + c_2^{k-1}c_2^k).
\]
If \( p_1 \) and \( p_2 \) have a common center at level \( i \), then the initial upper bound would be
\[
p_1p_2^L = (c_1^i + c_1^i + \cdots + c_1^{k-2}c_1^{k-1} + c_1^{k-1}c_1^i) + c_1^k + \\
(c_1^{k-1}c_2 + c_2^{k-1}c_2^{k-2} + \cdots + c_2^{k-1}c_2^k).
\]
A point \( p_3 \in D_3 \) should be opened only if
\[
p_1p_3 < p_1p_2^L - p_3p_2^L.
\]
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
\[
p_1p_3^U = p_1p_3 + (p_3c_1 + c_1^i + \cdots + c_1^{k-2}c_1^{k-1} + c_1^{k-1}c_1^i) + c_1^i + \\
(c_1^{k-1}c_2 + c_2^{k-1}c_2^{k-2} + \cdots + c_2^{k-1}c_2^k)
\]
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
\[
p_1p_3^U = p_1p_3 + (p_3c_1 + c_1^j + \cdots + c_1^{k-2}c_1^{k-1} + c_1^{k-1}c_1^j) + c_1^j + \\
(c_1^{k-1}c_2 + c_2^{k-1}c_2^{k-2} + \cdots + c_2^{k-1}c_2^k).
\]

\[C. \text{ Size and Effort Analysis}\]

We will first informally and then formally argue that the multilevel structure just presented is not a viable alternative to the two-level structure presented in Section II. The problem with the multilevel 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 storage would be smaller. However, we have additionally to 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 \( n = \prod_{i=1}^{k} d_i \). Note that in the case of two-level structure, we had only a \( d_0 \), which we called \( q \), and a \( d_1 \) which 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 \( O(d^k) \) 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 \( d >> 1 \), 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 \( O(n) + O(d^k) \). This expression is similar to the two-level case, the only difference being that \( d \) is likely to be significantly smaller now, since \( d = n^{1/k} \) for the multilevel 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 \( \delta_l \) through \( \delta_{k-1} \) be the maximum radii of the level 1 through level \( k-1 \) domains, and let \( \Delta_k = \prod_{i=1}^{k} \delta_i \). 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 \( \Delta_{k-1} \). 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 \( \Delta_{k-1} \) used as the radius. Thus, the effort required is bounded solely by the radius of the top-level domains. Since \( \Delta_{k-1} \) is large, we get very poor bounding with the multilevel structure.

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

\[D. \text{ An Embellishment}\]

We saw that the multilevel structure has better storage characteristics than the two-level structure but has poor bounding characteristics. We will now describe an embellishment of the above multilevel structure that inculces 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 II.

With this embellishment, the lower bound on distance between two points \( p_1 \) and \( p_2 \) is given by
\[
p_1p_2 \geq c_1^j + c_2^j - (c_1^{j-1}c_1^{j-2} + \cdots + c_1^j + c_1^i + i_2^1) + c_1^{j-1}c_2^j + \\
(c_1^{j-1}c_2^{j-1} + c_2^{j-1}c_2^{j-2} + \cdots + c_2^j + c_2^i),
\]
where \( i \geq k \) is the level at which there exists a domain such that the shortest distance between \( c_1^j \) and \( c_2^i \) has been stored.

The initial upper bound on the distance from points \( p_1 \) to \( p_2 \) is given by
\[
p_1p_2^U = (p_1c_1^j + c_1^j + \cdots + c_1^{j-2}c_1^{j-1} + c_1^{j-1}c_1^j) + c_1^{j-1}c_2^j + \\
(c_1^{j-1}c_2^{j-1} + c_2^{j-1}c_2^{j-2} + \cdots + c_2^j + c_2^i),
\]
where \( j \) is the smallest level at which the shortest distance between \( c_1^j \) and \( c_2^i \) 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
\[ p_1p_2' = p_1p_3 + (p_3c_1^1 + c_1^2 + \ldots + c_1^{l-1}c_1^{l-1} + c_1^l) \]
\[ + c_1^l + (c_2^{l-1} + c_2^{l-2} + \ldots + c_2^1 + c_2^2), \]
where $l$ is the smallest level at which the shortest distance between $c_1$ and $c_2$ 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 multilevel 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 V.

Before leaving the topic of the multilevel structures, we would like to make a few points in their favor. The reason the multilevel structure did so poorly in our analysis is that for large $k$, the top-level domains are very large and hence provide very weak bounds. The effectiveness of multilevel 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 multilevel structure may be a good choice. Finally, having a multilevel structure permits obtaining approximate solutions with great savings in effort, as discussed in the next section.

IV. APPROXIMATE SOLUTIONS—A SIDE BENEFIT

There may be situations in which an exact solution to the problem may not be necessary and an approximate solution with a bound on worst-case error may be satisfactory. A side benefit of the multilevel structure proposed in the previous section is that they can be used, albeit with some minor modifications, to derive successive approximations to the desired optimum until the desired accuracy has been achieved. In this section, we describe the modifications required in the multilevel structure and discuss how they can be used for obtaining successive approximations.

We require a few definitions. First is the concept of a neighbor. A level $j$ domain $D_2$ is said to be a $j$-neighbor of another level $j$ domain $D_1$ if and only if there exists a direct link from some node in $D_1$ to some node in $D_2$. The second concept is convexity. A domain is said to be convex if the shortest path between any two nodes in the domain passes only through nodes that are all members of the domain. (If there is more than one shortest path, at least one should satisfy this condition). We will insist in this section that each domain be convex.

The data structure now required with $k$ levels of domains is as follows: At the top level, compute and store the transitive closure of all level $k-1$ centers. At the next level, store the shortest distance and path from each level $k-2$ center, say $c$, to the center of a $k-2$ neighbor domain that is not in the same $k-1$ domain as $c$. Repeat, building a similar storage structure for each lower level. At the lowest level (0), the shortest path to neighbor requirement is automatically satisfied since these can only be direct arcs already present in the original graph. In addition, for each center (including level 0 centers, which are ordinary nodes), maintain the shortest path and center to the next higher level center.

With this data structure, the search procedure works as follows: Given the source node $s$ and the destination node $d$, first find the lowest level $j$ at which $s$ and $d$ belong to the same domain. A path from $s$ to $d$ is directly found by following a path from $s$ to the center of the domain so found, and then from the center to $d$. In the next step, the shortest path is found, using the techniques suggested in Section III-B, from the center of the level $j-1$ domain containing $s$ to the center of the level $j-1$ domain containing $d$, with the restriction that no nodes may be opened unless they are level $j-1$ domain centers. The path from $s$ to $d$ is constructed as the path from $s$ to its level $j-1$ domain center, the shortest path between the two level $j-1$ domain centers just found, and the path to $d$ from its level $j-1$ domain center. This step is then repeated at each lower level, each time using the length of the shortest path found at the previous level as an upper bound. One can either stop after one has a sufficiently good approximation or carry on until the exact shortest path is found.

Alternatively, one could decide in advance what level of accuracy and effort one desires, by means of an analysis such as the one given below. One could, for example, decide only to use one level, restricting consideration only to nodes that are level $k-1$ domain centers, and obtain an approximate answer as a straight look-up. Or one could decide to use $j$ levels, restricting consideration only to nodes that are domain centers at least at the $k-j$ level. The approximation to the shortest path is found as the path from the source node to its level $k-j$ domain center, the path computed between the two level $k-j$ domain centers, and the path to the destination from its level $k-j$ domain center.

A. Error, Effort, and Storage Analysis

Let $\delta_j$ represent the largest radius of a domain at level $j$, and let $\Delta_i$ be the effective radius such that the distance between the center of a level $i$ domain and its constituent nodes is no larger than $\Delta_i$, that is, $\Delta_i = \sum_{j=i}^{\infty} \delta_j$. Let the notation $c_i(p)$ represent the center of the level $i$ domain containing point $p$.

Then we have the following lemma.

Lemma 4.1:

\[ a_i(sd) \leq sd + 4\Delta_i \]
where \(sd\) represents the length of the shortest path from node \(s\) to node \(d\), and \(a_i(sd)\) is the length of an approximation to the shortest path by following a sequence of shortest paths from \(s\) to \(c_i(s)\), from \(c_i(s)\) to \(c_i(d)\), and finally from \(c_i(d)\) to \(d\).

**Proof:** We know that \(c_i(s)c_i(d) \leq sd + 2\Delta_i\), where \(c_i(s)c_i(d)\) is the length of the shortest path from node \(c_i(s)\) to node \(c_i(d)\). (This must be the case since we can always take a path from \(c_i(s)\) to \(s, d\) and then \(c_i(d)\) that is no more than \(sd + 2\Delta_i\) long.) In addition, the shortest paths from \(s\) to \(c_i(s)\) and from \(c_i(d)\) to \(d\) are each no more than \(\Delta_i\) long, by definition of \(\Delta_i\). Adding these up, we have the desired result for path \(a_i(sd)\).

In words, Lemma 4.1 states that an approximation to the shortest path between nodes \(s\) and \(d\) using the shortest path between their respective level \(i\) centers is guaranteed to be within \(4\Delta_i\), of the true shortest path. Since \(\Delta_i\) increases with \(i\), the more the number of levels considered (the lower the \(i\)), the lower will be the error.

Notice that the error bound is a function only of the domain sizes and is independent of the actual distance between the two nodes specified in the query. In other words, the proportional error is likely to be quite small when the two points given are far apart and could be high if the points given are close together. However, when points are close together, the effort required for an exact search is not large and a multilevel organization can be used to prune the search effectively. Thus, with a multilevel data structure including neighbor edges, one could choose the level at which to conduct a search depending upon the magnitude of the distance between the points in question, more levels being included when the points are close together and only a few levels being considered when the points are far apart.

Our approximation procedure differs from most heuristics in that while the latter can be used to efficiently find solutions close to the optimum, there is usually no bound on the worst-case error, and hence one can never be certain of how good the solution obtained is. In our technique, even though the actual performance may be considerably better, specific upper bounds can be placed on the error margins.

In terms of effort, if search is restricted to go up only to the \(i\)th level from the bottom, one has effectively to work with a graph that consists only of nodes that belong to top \(k - i\) levels, which is only \(n/d^i\) nodes. Recall that there are in all \(n\) nodes in the graph, and each domain has \(d\) points. The point to note is that as \(i\) decreases or, equivalently, \(k - i\) increases (that is, more levels are examined), the effort required increases exponentially. However, the approximation to the shortest distance also improves. This tradeoff can be made as appropriate for each query posed to the database.

The storage requirement is approximately the same as in the ordinary multilevel structure, except that each domain center at levels 1 through \(k - 2\) now has roughly \(g\) additional edges as a result of maintaining paths to neighbor domains. (At level 0, the edges to neighbors are a subset of the original edges, and at level \(k - 1\) a complete closure is being maintained.) The domains at level 1 dominate this storage and are \(d^{k-1} = n/d\) in number. Therefore, the number of additional edges is \(O(g n/d)\). Note that \(g\) is expected to be a single-digit number.

V. 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.

A. 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 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 [14]. One would like to make the effort ratio as close to 0 as possible.

Following [5], 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 3, given in Section II-F, was used to divide the nodes into various domains. Most of the experiments were performed with a graph of 2500 nodes, with average outdegree of 8 and 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.
IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 6, NO. 2, APRIL 1994

B. 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. Fig. 5 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 becomes 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 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. Fig. 6 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.

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

C. 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 II-E. The effort and size ratios have been plotted in Fig. 7 as a function of domain size for the same 2500-node database.

In Fig. 7, the shape of the effort ratio curve is the same as in Fig. 5 for the two-level structure without domain closures, except for a slight reduction in the effort required. This is not very surprising, given our observation in Section II-E that no significant improvement in bounding is obtained by keeping the domain closures.

However, the size ratio curve in this figure has an interesting shape. When the domain sizes are small, little storage is required to store the domain closures. There are a large number of domain centers, and the total storage requirement is dominated by the closure between these domain centers, and the curve follows a downward trend similar to Fig. 5. But for large domain sizes, the domain closures become large and dominate the storage requirement, causing the size ratio to increase strongly with increasing domain size. As discussed in Section II-E, the optimum domain size is of the order of the cube root of the number of nodes.

Fig. 8 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 Figs. 5 and 7. Note from Fig. 7 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 Fig. 8.

It is apparent from Fig. 8 that performancewise 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 structure without domain closures incurs less storage overhead. Although they are not presented here, similar results were 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.

D. Experiment 3: Multilevel Structure

The third set of experiments examined the effectiveness of the multilevel structure with domain closures presented in Section III. 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. Fig. 9 shows the effort and size ratios for different number of levels in the multilevel structure.

For two levels, the multilevel 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 III, 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 versus size curves for the two-level structure and the multilevel structure in Fig. 10. It can be seen that performancewise, for most of the operating region, the two-level structure dominates the multilevel structure. However, it is possible to reduce the storage overhead at a level that is not possible with the two-level structure at considerable loss in speed-ups. Note that the two-level structure requires at least twice the number of nodes units of additional storage (to store distance from each node to its domain center and vice versa), whereas the multilevel structure one could go below this bound on storage overhead.

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

E. Experiment 4: Approximate Solutions

The final set of experiments examined the effectiveness of the approximate solution scheme presented in Section IV. Recall that an approximate solution requires a multilevel structure, and the search is restricted to $j$ top levels, depending upon the accuracy desired. The experiments were performed with the 2500-node database with a six-level structure. The results are presented in Fig. 11.

In this graph, levels represent the number of levels from the top that were considered in determining the shortest path. Thus, for levels $= 1$, only the top level was examined, and for levels $= 6$, all levels were examined. Effort ratio is as defined before. The distance ratio is defined to be the ratio of the approximate value of the shortest distance divided by its exact value, and is a measure of error in approximation.

As the number of levels up to which search is performed increases, the approximation error reduces, but at the same time the effort ratio also increases. Notice, however, that the
effort ratio increases very slowly in the beginning. Therefore, one may obtain very good approximations expending very little effort using this approach. In fact, the graph plotted in the figure is averaged over a large number of pairs of points, some close together and some far apart. While the graph is plotted over the entire set of points for consistency with the other graphs presented in this section, a few pairs of points that lie close together have very poor distance ratios (recall that the absolute error could be as high for a pair of nearby points as for a pair of far-away points), and greatly bias the average ratio upwards. With these pairs of points disregarded, even with only one level considered (direct look-up in our data structure), the approximate solution obtained had an average distance ratio of around 1.6.

F. 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 multilevel 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 multilevel 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 multilevel structure one could go below this bound on storage overhead. An attractive use of multilevel structure is in obtaining approximate solutions. If one can live with an approximate solution rather than the exact solution, a much larger speed-up can be realized using a multilevel structure as opposed to a two-level structure. With a multilevel structure, one could also choose the desired level of accuracy and obtain successive approximations with increasing amounts of work.

VI. GENERALIZATIONS

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

A. 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 I, 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, p1 and p2, 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

\[ p_1p_3 + p_3p_2^U > p_1p_2^L. \]

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 [26] procedure. Note that the seminaive 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 \( (\text{node}, \text{distance}) \), 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, \( pq_L \)

\begin{verbatim}
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 \( j \in \text{Succ}(i) \) 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. (6.1)
      if \( j \) needs to be opened then
        append \( (j, d_i + d_{ij}) \) to OPEN;
        update \( pq_L \)

\}
\end{verbatim}

\subsection*{B. Multiple Domain Membership}

We have thus far assumed that the nodes have been divided into nonintersecting domains, so that each node has a unique domain center. If a node is allowed to belong to more than one domain, 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.

\subsection*{C. Incremental Changes}

Whenever some derived information is materialized, a change in the base information must be reflected in a change in the derived information [9], [15], [18], [28]. 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 must be updated. Obviously, a complete recomputation would be extremely expensive. One possibility is to use the incremental techniques suggested in [3]. However, given the extremely large sizes of graphs that we have in mind, even these incremental techniques may be too expensive to use frequently. We describe below an alternative approach.

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 could 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. In consequence, 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 have to be recomputed. The optimal point at which to carry out such a recomputation is an open research problem. Our claim here is that in a quasistatic situation, the approach suggested here can become quite attractive, in spite of the additional storage overhead and complexity that it introduces.

\section*{VII. CONCLUSION}

In this paper, we considered the problem of performing efficient search over disk-resident massive graphs. 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 branch and bound search algorithms used in many expert systems, and new algorithms can be designed using our bounding procedure. We emphasize that although in this paper we have used the shortest path problem as the running paradigm and Dijkstra's algorithm for finding the shortest path as the search algorithm in which the bounding procedure was incorporated, it was primarily done to keep the discussion concrete, and our techniques can easily be adapted to solve any number of useful and practical problems such as the shortest path, critical path, largest capacity path, path of maximum reliability, etc. [1], [5], [8], [10]-[12], [27], [29] Simulation results confirm that, using these techniques, a search can be expedited significantly without incurring a large storage penalty. A side benefit of this work is that we have shown how search can be organized to obtain successive approximations to the desired solution with considerable reduction in effort.

\section*{REFERENCES}

Rakesh Agrawal received the M.S. and Ph.D. degrees in computer science from the University of Wisconsin, Madison, in 1983. He also received the B.E. degree in electronics and communication engineering from the University of Roorkee, Roorkee, India, and the two-year Postgraduate Diploma in industrial engineering from NITIE, Bombay, India. From 1983 to 1989 he was with AT&T Bell Laboratories, Murray Hill, NJ, where he was a member of the technical staff in the Computing Systems Research Laboratory. Since January 1990 he has joined the Computer Science Department of the IBM Almaden Research Center, San Jose, CA. His current research interests include database programming languages, object-oriented database systems, deductive databases, and data mining.

H. V. Jagadish received the Ph.D. degree from Stanford University, Stanford, CA, in 1985. Since then he has been with the Computing Systems Research Center at AT&T Bell Laboratories, Murray Hill, NJ. His current research interests include object-oriented database systems, multimedia information management, data stream processing, and databases for networks.