Storing a Collection of Polygons Using Quadtrees

HANAN SAMET
University of Maryland
and
ROBERT E. WEBBER
Rutgers University

An adaptation of the quadtree data structure that represents polygonal maps (i.e., collections of polygons, possibly containing holes) is described in a manner that is also useful for the manipulation of arbitrary collections of straight line segments. The goal is to store these maps without the loss of information that results from digitization, and to obtain a worst-case execution time that is not overly sensitive to the positioning of the map. A regular decomposition variant of the region quadtree is used to organize the vertices and edges of the maps. A number of related data organizations are proposed in an iterative manner until a method is obtained that meets the stated goals. The result is termed a PM (polygonal map) quadtree and is based on a regular decomposition point space quadtree (PR quadtree) that stores additional information about the edges at its terminal nodes. Algorithms are given for inserting and deleting line segments from a PM quadtree. Use of the PM quadtree to perform point location, dynamic line insertion, and map overlay is discussed. The PM quadtree is compared conceptually to the K-structure and the layered dag with respect to typical cartographic data. An empirical comparison of the PM quadtree with other quadtree-based representations for polygonal maps is also provided.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*trees*; I.2.10 [**Artificial Intelligence**]: Vision and Scene Understanding—*representations, data structures, and transforms*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*curve surface, solid, and object representations*; *geometric algorithms, languages, and systems*; *modeling packages*; I.4.6 [**Image Processing**]: Segmentation—*region growing, partitioning*; I.4.7 [**Image Processing**]: Feature Measurement—*size and shape*

General Terms: Algorithms

Additional Key Words and Phrases: Geographic information, hierarchical data structures, line representations, map overlay, polygonal representations, quadtrees

1. INTRODUCTION

Hierarchical data structures are becoming increasingly important representation techniques in the domains of computer graphics, image processing, computational geometry, geographic information systems, and robotics. They are based on the principle of *divide and conquer*. One such data structure is the *quadtree* [12, 19].

This work was supported by the National Science Foundation under grant DCR-83-02118.
Authors' addresses: H. Samet, Computer Science Department, University of Maryland, College Park, MD 20742; R. E. Webber, Computer Science Department, Rutgers University, Busch Campus, New Brunswick, NJ 08903.

It is our goal to show how one variant of the quadtree data structure can be adapted to the problem of representing polygonal maps. We define a *polygonal map* as a collection of polygons (bounding disjoint polygonal regions) which may be degenerate (e.g., isolated vertices or edges). Thus a polygonal map is an arbitrary collection of line segments that only intersect at vertices and that implicitly define a collection of bounded regions. The implicitly defined regions may contain holes. Such a map must be embedded in a Euclidean plane. These maps are sometimes referred to as planar subdivisions and frequently arise in cartographic applications (e.g., representing county boundaries, etc.) as well as in a host of other applications that require partitioning of the plane into regions whose boundaries consist of straight line segments. In our discussion, we use the terms line segment, line, and edge interchangeably, and likewise for the terms point and vertex.

Our presentation is an evolutionary one (along the lines first presented in [17, 23]) in the sense that we start with one data structure and continually refine it until it satisfies our goal. Although this approach has some merit as a case history of the design of a data structure, its main purpose is to show the possible trade-offs involved with different criteria for leaf determination. When storing one-dimensional data, we have practical, optimal, general-purpose data structures, that is, the 2–3 tree implementation of a dictionary [1]. However, when manipulating two-dimensional data, the situation is more complicated. Some practical heuristics, like the quadtree decomposition paradigm, perform well on typical data, but poorly for various cases of extremely skewed data. Other, more robust techniques (e.g., the K-structure [11] and the layered dag [5] discussed in Appendix 2) appear to be less suited to the manipulation of real data.

The remainder of this paper is organized as follows. In Section 2 we review the concept of a quadtree and discuss some of its variants in the context of the type of data we wish to represent. In Section 3 we develop the PM quadtree—our solution to the problem of storing a polygonal map. Section 4 outlines some problems that can be solved using the PM quadtree including the actual mechanics of their solution. In particular, it is shown how to perform point location, dynamic line update (although we only discuss the analysis of dynamic line insertion), and map overlay. Section 5 is an empirical comparison of the storage requirements of the PM quadtree and other quadtree-based data structures when used to encode some sample cartographic data. Three appendices are included to handle some peripheral topics. Appendix 1 contains a set of procedures for the insertion and deletion of line segments in the PM quadtree. Appendix 2 reviews other solutions to the problems discussed in Section 4 but that are not based on quadtrees as well as a comparison of their use with solutions based on the PM quadtree. Appendix 3 discusses how labels of regions are maintained so that region identification information can be updated efficiently when line segments are inserted and deleted.

The point location problem is to determine the region that contains a given pair of coordinates. Since we do not represent the region explicitly, this problem reduces to locating a line segment that borders the region that contains the point (and knowing on which side of the line segment the region lies). This problem is central to the input aspects of interactive computer graphics. The dynamic line insertion problem is the problem of adding new information (in the form of new

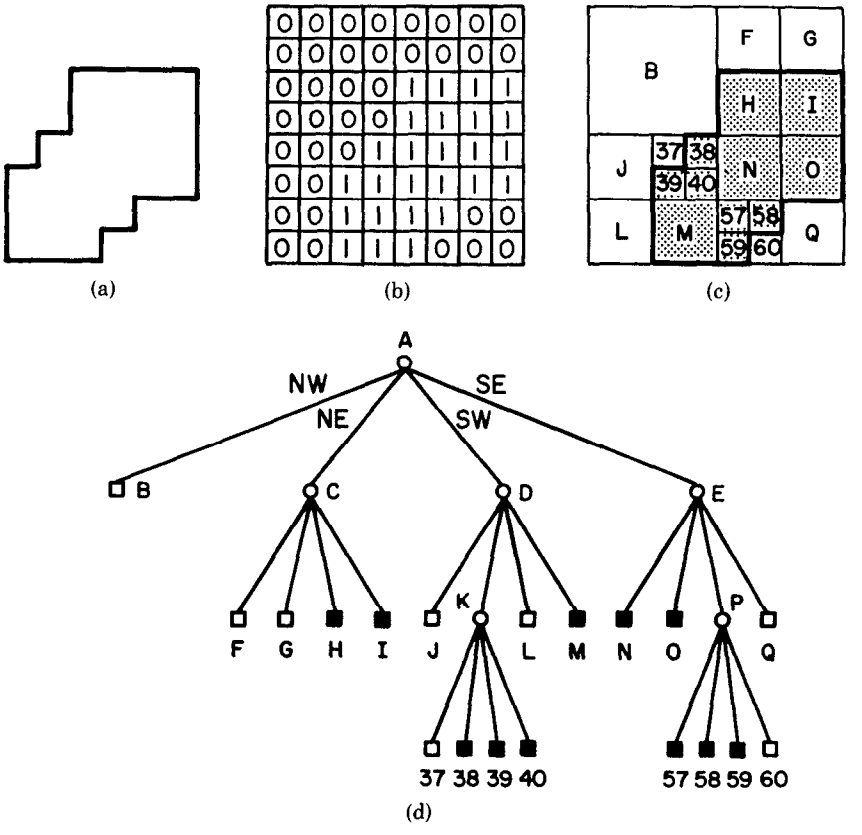


Fig. 1. A region, its binary array, its maximal blocks, and the corresponding quadtree. (a) Region. (b) Binary array. (c) Block decomposition of the region (a). Blocks in the region are shaded. (d) Quadtree representation of the blocks in (c).

line segments) to an already existent structure. This problem is central to the real-time aspects of interactive computer graphics. The map overlay problem can be viewed as a generalization of the dynamic line insertion problem in that instead of inserting a single line, we are now inserting a collection of lines (which happen to be already organized as a map). Our technique will be seen to be more efficient than performing a sequence of dynamic line insertions.

2. OVERVIEW OF QUADTREE DATA STRUCTURES

2.1 Region Quadtrees

In its most general context, the term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition [19]. In two dimensions, a square planar region is recursively subdivided into four rectangular parts until each part contains data that is sufficiently simple so that it can be organized by some other data structure (e.g., a vector or a dictionary). They can be differentiated on the basis of the type of data that they are used to represent, and on the principle

guiding the decomposition process. The decomposition may be into congruent parts of the same shape as the original planar region (termed a *regular decomposition*) or it may be governed by the input.

As an example of the quadtree concept, we briefly indicate how it is used to represent digitized regions. Consider the region shown in Figure 1a which is represented by the 2^3 by 2^3 binary array in Figure 1b. Observe that the 1's correspond to picture elements (termed *pixels*) that are in the region and the 0's correspond to pixels that are outside the region. The most studied quadtree approach to region representation, termed a *region quadtree*, is based on the successive subdivision of the array into four equal-size quadrants. If the array does not consist entirely of 1's or entirely of 0's (i.e., the region only partially overlaps the entire array), then we subdivide it into quadrants, subquadrants, ..., until we obtain blocks (possibly single pixels) that consist entirely of 1's or entirely of 0's; that is, each block is entirely contained in the region or entirely disjoint from it. As an example, the resulting blocks for the array of Figure 1b are shown in Figure 1c. This process is represented by a tree where each nonleaf node has four sons. The root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE). The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be BLACK or WHITE depending on whether its corresponding block is entirely inside or entirely outside of the represented region. All nonleaf nodes are said to be GRAY. The quadtree representation for Figure 1c is shown in Figure 1d.

2.2 PR Quadtrees

Point data can be represented by a point quadtree [6] which is a decomposition of a square planar region into noncongruent parts. The region quadtree can also be adapted to represent point data. We term such a tree a PR quadtree (PR denoting point region) [15]. PR quadtrees store points only in terminal nodes. Regular decomposition is applied until no quadrant contains more than one data point. For example, Figure 3 is a PR quadtree representation of the five vertices of the polygonal map in Figure 2. An interesting problem arises when vertices lie on the border of quadtree nodes. We could always move the vertices so that this does not happen, but generally this requires global knowledge about the maximum depth of the quadtree prior to its construction. We could also establish the convention that some sides of the region represented by a node are closed and other sides are open, but this can lead to implementation difficulties when floating point numbers are involved. Therefore, we adopt the convention that all sides of a node are closed. This means that a vertex that lands on the border between 2 (or 3 but never more than 4) nodes is recorded in each of the nodes on whose border it exists.

When comparing the PR quadtree to the point quadtree, three distinctions are worthy of note. First, the PR quadtree is algorithmically simpler than the point quadtree. This is particularly noticeable in the implementation of a point deletion procedure.

Second, given a subtree of a PR quadtree there is an upper bound on the separation, in terms of distance, between its points. For example, assuming that

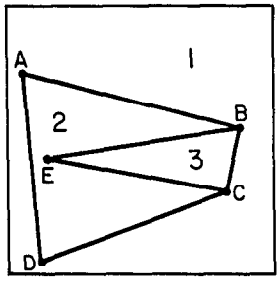


Fig. 2. Sample polygonal map.

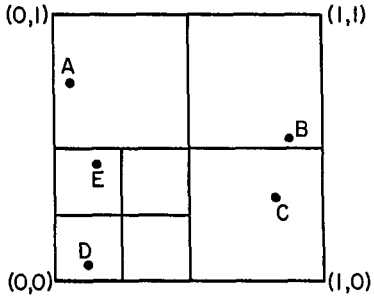


Fig. 3. PR quadtree corresponding to the vertices A, B, C, D, and E, of polygonal map of Figure 2.

the universe is the unit square, a subtree whose root is at a depth d will represent a 2^{-d} by 2^{-d} region. Thus, the maximum separation between any two points in it is bounded from above by $\sqrt{2}/2^d$. In contrast, the bound for the point quadtree is $\sqrt{2}$.

The third distinction is in their balance, that is, the distribution of the depths at which the leaf nodes are found. In the case of the PR quadtree, this is a function of the static distribution of the points in the area of concern. If the points are uniformly distributed, then they can be expected to be stored in leaves with a depth of $\log_4 v$ where v is the number of points. When the points result from a curve sampling process (e.g., the vertices of a polygonal approximation of a boundary), then as the density of the sampling process increases, the points will be stored in leaves whose depth is bounded from above by $\log_2 v$. However, in the worst case (i.e., for skewed point distributions), the only upperbound available is a function of the closest approach between the points (as discussed in the next section). In contrast to this, the balance of the point quadtree is a function of the order in which the points are inserted. A simple algorithm that sorts the points by one coordinate and then uses medians for roots of subtrees can build a point quadtree with a maximum depth of $\log_2 v$. Overmars and van Leeuwen [16] show that by performing this sort algorithm whenever the balance becomes worse than some constant multiplicative factor, an overall performance of $O(v \cdot \log_2 v)$ for the insertion of v points can be maintained. This approach is not suitable for an interactive environment, however, because it entails an

occasional $O(v \cdot \log_2 v)$ worst-case insertion cost for inserting only one point into a map of v points.

The above considerations lead us to conclude that among the quadtree-based data structures, the PR quadtree forms a better basis than the point quadtree for the development of a practical algorithm to process typical cartographic data in an interactive environment. Alternative representations based on quadtrees are discussed briefly below. The PR quadtree is compared with nonquadtree alternatives in Appendix 2.

2.3 Previous Approaches to Storing Line Data in Quadtrees

There are two major approaches to the representation of regions: those that specify the borders of a region and those that organize the interior of a region. This corresponds to either storing region identification information only on the region's border or also storing it on parts of the region's interior. The definition of the region quadtree given above corresponds to the latter approach. In the case of polygons, we are more interested in the former approach. Hunter and Steiglitz [9, 10] address the problem of representing simple polygons (i.e., polygons with nonintersecting edges and without holes) with a region quadtree which, when used this way, we call an MX quadtree. A disadvantage of the MX quadtree is that shift and rotate operations may result in information loss with respect to the map that was originally digitized.

Some alternative but related approaches include the edge quadtree [13, 21], edge-EXCELL [22], and the line quadtree [18]. These are not suitable for polygonal maps due to, in part, difficulties in representing an arbitrary number of vertices. The regular decomposition property of the region quadtree is very important because it enables the efficient execution of set-theoretic operations such as union and intersection of two regions, polygons, etc. This results from the quadtrees being in registration and enables uninteresting areas to be ignored by virtue of the hierarchical representation. For the region quadtree, these operations can be performed in time proportional to the number of nodes in the quadtrees involved [9, 10]. In the following we examine the edge and MX quadtrees more closely as we will use them as a benchmark against which to evaluate the PM quadtree—our proposed data structure.

In the *edge quadtree* of Shneier [21] a region containing a vector feature, or part thereof, is repeatedly subdivided into subquadrants until each quadrant contains a curve that can be approximated by a single straight line segment. Each leaf node contains the following information about the edge passing through it: magnitude (i.e., 1 in the case of a binary image or the intensity in case it is a grey-scale image), direction, intercept, and a directional error term (i.e., the error resulting from the approximation of the curve by a straight line using a measure such as least squares). If a line segment terminates within a node, then a special flag is set and the intercept denotes the point at which the segment terminates.

Applying this process leads to quadtrees in which long straight edges can be stored in a few large leaves. However, small leaves are required in the vicinity of corners, intersecting edges, close approaches between curves, or areas of high curvature. Of course, many leaves will contain no edge information at all, since

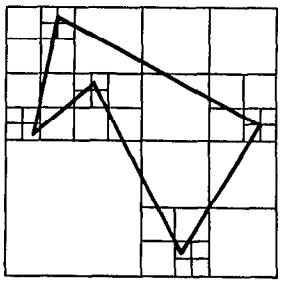


Fig. 4. Sample edge quadtree.

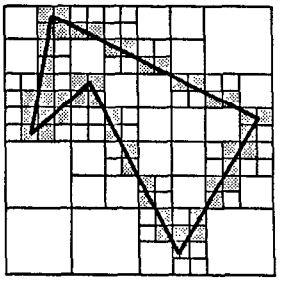


Fig. 5. MX quadtree corresponding to Figure 4.

they are not intersected by a curve. As an example of the decomposition that is imposed by the edge quadtree, consider Figure 4 which is a sample polygon and its corresponding edge quadtree when represented on a 2^4 by 2^4 grid.

A serious drawback of the edge quadtree is its inability to handle the meeting of two or more edges at a single point (i.e., a vertex) except as a pixel corresponding to an edge of minimal length. The problem is that at a certain level of decomposition all vertices are represented by single line segments regardless of their degree. This means that boundary following as well as deletion of line segments cannot be properly handled in the vicinity of a vertex at which more than one edge meets.

The *MX quadtree* [10] is closely related to the edge quadtree. It considers the border of a region as separate from either the inside or the outside of that region. Figure 5 shows the MX quadtree corresponding to the polygon of Figure 4. The MX quadtree has problems similar to those of the edge quadtree in handling vertices. Again, a vertex is represented by a single pixel. Thus boundary following and deletion of line segments cannot be properly handled. Worse is the fact that an MX quadtree only yields an approximation of a straight line rather than an exact representation as done by the edge quadtree. This leads to faster deterioration of accuracy when the representation is rotated. Moreover, the MX quadtree usually contains more nodes than the edge quadtree as can be seen by comparing Figures 4 and 5.

3. THE DEVELOPMENT OF THE PM QUADTREE

The quadtree that we develop for storing polygonal maps will be referred to as the PM quadtree. It will be seen to be an adaptation of the PR quadtree. Our goal in designing this data structure is to derive a reasonably compact represen-

tation that satisfies the following three criteria:

(1) It stores polygonal maps without information loss (i.e., it does not suffer a loss of accuracy resulting from digitization).
(2) It is not overly sensitive to the positioning of the map (i.e., shift and rotation operations do not drastically increase the storage requirements of the map).
(3) It can be efficiently manipulated.

To meet these goals, we develop three closely related quadtree structures: PM_1, PM_2, and PM_3. Our approach is to find a decomposition criterion that corresponds to the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure. We find this decomposition criterion by successively weakening the definition of what constitutes a permissible leaf node thereby enabling more information to be stored at each leaf node. Thus a permissible PM_1 quadtree leaf node is also a permissible PM_2 quadtree leaf node and likewise a permissible PM_3 quadtree leaf node.

3.1 General Remarks

In general, it is difficult to evaluate a data structure without some operations in mind. We are interested in performing interactive computer graphics on cartographic data (in particular, maps represented as a collection of line segments). Thus, we evaluate the PM quadtree in the context of the following three tasks: point location, dynamic line insertion (algorithms for the more general dynamic update task are given in Appendix 1), and map overlay. We assume that the polygonal map is being manipulated in a dynamically changing environment. We also assume that associated with each line segment that forms the boundary of a region, there is a pair of names indicating which region is on which side of the line segment. This reduces the point location task to the less restricted problem of locating a line segment that borders the region containing the query point. For example, the map of Figure 2 partitions the plane into 3 regions, labeled 1, 2, and 3. Thus, for this figure, edge DA is marked to indicate that region 2 lies on the right-hand side of DA and region 1 lies on the left-hand side of DA. A discussion of how such labels are maintained is orthogonal to our presentation of the development of the PM quadtree and is therefore treated in Appendix 3.

The quadtrees presented in this section will be defined in terms of their decomposition criteria. First, let us consider the definition of a PR quadtree in terms of its criterion for decomposing a quadrant. The decomposition criterion that defines the PR quadtree is termed C1 and is given below:

C1: At most one vertex can lie in a region represented by a quadtree leaf.

Figure 3 shows the PR quadtree formed from the vertices of the map of Figure 2.

The analysis of each of the PM quadtree variants relies heavily on the value of the worst-case tree depth, which is, in turn, a function of the input polygon—that is, the minimal angle formed by adjacent edges in the polygon and the closest approach of various parts of the polygon. Thus, it is worthwhile to first analyze the depth of the PR quadtree. The worst-case PR quadtree depth is obtained as

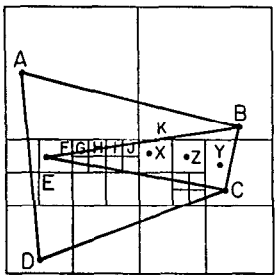


Fig. 6. PM_1 quadtree meeting criteria (C1, C2′, and C3).

follows. Assume that the polygonal map is embedded in a unit square. As the depth of the PR quadtree increases, the maximum separation between two points in the same node is halved. The maximum separation between any two points in the unit square is $\sqrt{2}$. Points that are this far apart require a tree with depth 1 to separate them. Generalizing this observation, we see that letting $d_{\min.vv}$ be the minimum separation between two distinct vertices, then an upperbound on the depth of the corresponding quadtree is

$$\mathrm{D1} = 1 + \log_2 \frac{\sqrt{2}}{d_{\min.vv}}.$$

In the subsequent discussion, we frequently need to refer to segments of edges of the polygonal map (for which we also use the term straight-line planar graph or a suitable abbreviation thereof) that are formed by clipping an edge of the polygonal map against the border of the region represented by a quadtree node. We use the term *q-edge* (denoting a quadtree-decomposition edge) to refer to such an edge (e.g., EF and FG in the quadtree decomposition of Figure 6). Thus every map edge is covered by a set of q-edges that only touch at their endpoints. For example, edge EB in Figure 6 consists of the q-edges EF, FG, GH, HI, IJ, JK, and KB. Note that only B and E are vertices; F, G, H, I, J, and K merely serve as reference points.

3.2 The PM_1 Quadtree

A criterion analogous to C1, called C2, which takes edges into account is given below.

C2: At most one q-edge can lie in a region represented by a quadtree leaf.

It should be clear that C2 does not imply C1 due to the possible presence of isolated vertices. Nevertheless, C2 is inadequate because there exist polygonal maps that would require a PM quadtree of infinite depth to satisfy C2. For example, consider the portion of such an infinite decomposition shown in Figure 6. The node containing vertex E does not satisfy C2 because of the two q-edges incident at it. Assume that the x and y coordinates of E cannot be expressed (without error) as a rational number whose denominator is a power of two (e.g., let both coordinates be $\frac{1}{3}$). This means that E can never lie on the boundary between two quadrants. Thus, by virtue of the continuity of the q-edges, no matter how many times we subdivide the quadrant containing vertex E, there

will always exist a pair of (possibly infinitesimally small) q-edges incident at E which will occupy the same quadtree leaf.

One solution to the above problem lies in replacing C2 with criteria C2′ and C3 given below.

C2′: If a region contains a vertex, then it can contain no q-edge that does not include that vertex.

C3: If a region contains no vertices, then it can contain at most one q-edge.

A quadtree built from the criteria C1, C2′, and C3, representing the polygonal map of Figure 2, termed a *PM_1 quadtree*, is shown in Figure 6. Note that as in the PR quadtree, when a vertex lands on the border between 2 or 3 or 4 nodes, then it is inserted in all of the nodes on whose border it exists. By the same reasoning, when a line or q-edge falls on the border between two quadrants, it is inserted in all of the nodes on whose border it exists. Thus we say that all four quadrants are closed.

Since criterion C2′ allows any number of q-edges to be stored at one PM_1 quadtree leaf, a question arises as to how these q-edges are organized. The simplest approach, consistent with our interest in worst-case tree-depth, is to store the q-edges in a dictionary [1] where the q-edges are ordered by the angle that they form with a ray originating at the vertex and parallel to the positive x axis. For efficient updating and search, the dictionary itself is usually implemented as some type of a tree structure such as a 2–3 tree [1] (but not as a quadtree since the dictionary is storing a linear ordering). Since the number of q-edges passing through a leaf is bounded from above by the number of vertices belonging to the polygonal map, say V, the depth of the dictionary structure is at most

$$\text{A1} = \log_2 V + 1.$$

The depth of the PM_1 quadtree can be determined as the maximum of the depth required independently by each of the criteria (i.e., C1, C2′, and C3) for building the quadtree. The factor contributed by criterion C1 has already been noted to be D1. If $d_{\text{min.}ev}$ denotes the minimum separation between an edge and a vertex not on that edge (for a given polygonal map), then by reasoning similar to the derivation of D1, the depth of the PM_1 quadtree required to fulfill criterion C2′ is

$$\text{D2}' = 1 + \log_2\left(\frac{\sqrt{2}}{d_{\text{min.}ev}}\right).$$

Note that a map consisting of a single line segment would have no meaningful value for $d_{\text{min.}ev}$.

Analogously, if $d_{\text{min.}ee}$ denotes the minimum separation between two nonintersecting q-edges (i.e., portions of edges bounded by either a vertex or the boundary of a PM_1 quadtree leaf of the PM_1 quadtree of the given polygonal map), then the PM_1 quadtree depth required to fulfill criterion C3 is

$$\text{D3} = 1 + \log_2\left(\frac{\sqrt{2}}{d_{\text{min}.ee}}\right).$$

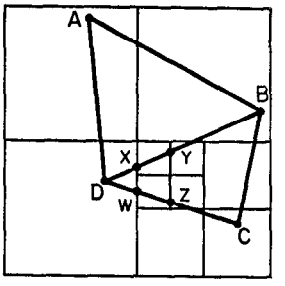


Fig. 7. Example illustrating D3 > D2′ when C3 is used in a PM_1 quadtree.

The factors D1 and D2′ are functions of the polygonal map and are independent of the positioning of the underlying digitization grid. However, the factor D3 is dependent on the positioning of the digitization grid and thus it can vary as the polygonal map is shifted. Recall that each of these factors, D1, D2′, and D3, is an upperbound on some aspect of the quadtree's construction that could contribute to the depth of the resulting quadtree. The actual depth of the quadtree that is built could be considerably less than any of these factors. For maps of the complexity of the one shown in Figure 2, the D3 factor can become arbitrarily large. For example, suppose we shift the polygonal map in Figure 2 to the right. As vertex E (see Figure 6) moves closer and closer to the quadrant boundary on its right, the minimum separation between the q-edges of BE and CE that are not incident at E becomes smaller and smaller resulting in the growth of D3 to unacceptable values. While the PM_1 quadtree does not have the problem associated with the use of C2, it still behooves us to find a better decomposition criterion than C3 because we would like to be able to represent an image with a fixed amount of storage irrespective of the positioning of the underlying digitization grid.

3.3 The PM_2 Quadtree

In order to remedy the deficiency associated with criterion C3, it is necessary to determine when it dominates the cost of storing a polygonal map. In particular, D3 is greater than D2′ only if $d_{min.ee}$ is smaller than $d_{min.ev}$, which happens only when the two nearest nonintersecting q-edges are segments of edges that intersect at a vertex. For example, Figure 7 is the PM_1 quadtree for a polygonal map ABCD, such that D3 is greater than D2′ because $d_{min.ee}$ (the distance between q-edges XY and WZ) is smaller than $d_{min.ev}$ (the distance between C and BD). Note that XY is a q-edge of BD, WZ is a q-edge of CD, and BD intersects CD at vertex D. This analysis leads us to replace criterion C3 with criterion C3′ defined below.

C3′: If a region contains no vertices, then it can contain only q-edges that meet at a common vertex exterior to the region.

A quadtree built from criteria C1, C2′, and C3′, for the polygonal map of Figure 2, termed a *PM_2 quadtree*, is shown in Figure 8. Note that q-edges in the same leaf meeting criterion C3′ would be ordered angularly in a dictionary as were those q-edges meeting criterion C2′.

The worst-case tree depth is again proportional to the sum of the depth of the quadtree plus A1, the maximum depth of the dictionary structure. However, the

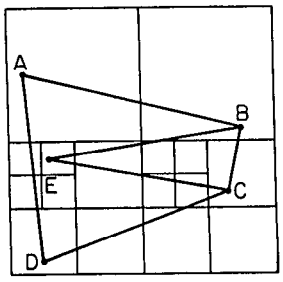
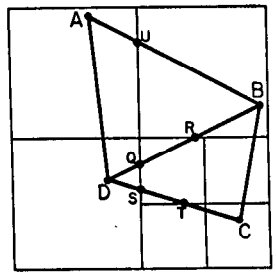


Fig. 8. PM_2 quadtree meeting criteria C1, C2′, and C3′.

Fig. 9. The PM_2 quadtree of Figure 7 when C3′ is used instead of C3.



depth of the quadtree is bounded from above by the maximum of D1 and D2′, the factors attributed to criteria C1 and C2′, respectively. Note that by virtue of our definition of C3′, the maximum depth resulting from its use is bounded from above by D2′.

As an example, consider Figure 9, which represents the same polygonal map as Figure 7 except that it uses C3′ instead of C3. The analog of $d_{\text{min.}ee}$, termed $d_{\text{min.}ee'}$, is defined as the minimum separation between two q-edges that are not segments of two intersecting edges. In this example, D3′ is less than D2′ because $d_{\text{min.}ee'}$ (the distance between UB and SC) is greater than $d_{\text{min.}ev}$ (the distance between C and DB). Note that the distance between QR and ST, and the distance between RB and TC, are irrelevant to D3′, because, if necessary, these segments could be in the same leaf.

We have now found a criteria for which the worst-case tree depth is less sensitive to shift and rotation of the polygonal map. The only question that remains is whether we can do better. Can the contribution of criterion C2′ be removed or reduced?

3.4 The PM_3 Quadtree

We consider a quadtree, termed a *PM_3 quadtree*, which is built using only criterion C1, but that could represent any polygonal map. For example, Figure 10 is the PM_3 quadtree corresponding to the polygonal map of Figure 2. Recall that the PR quadtree is also built using only criterion C1. Thus the number of quadtree nodes in the PR quadtree for the vertices of a polygonal map is equal to the number of quadtree nodes in the PM_3 quadtree of the polygonal map, although the amount of information stored in the quadtree leaf node of a PM_3 quadtree

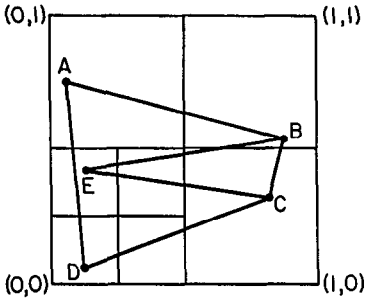


Fig. 10. PM_3 quadtree corresponding to the polygonal map of Figure 2.

can be much greater than the amount of information stored in the quadtree leaf node of a PR quadtree. Since the depth factor D1 is always less than or equal to the maximum of the factors D1 and D2′, the quadtree component (as opposed to the dictionary component) of the worst-case tree depth is lower than in our previous structures. Indeed, the only time D1 is greater than D2′ is when the polygonal map contains isolated edges (i.e., edges with both endpoints of degree 1). If the isolated edges were short, then $d_{\min.vv}$ would be small. But, if the isolated edges were far apart, then $d_{\min.ev}$ would be large. However, this structure does have the problem that the number of q-edges that can be stored in a leaf is now bounded by the number of edges in the graph, instead of the number of vertices. This does not affect the order of the worst-case tree depth, because, in a planar graph (containing neither multiple edges nor nonlinear edges), the number of edges is bounded from above by six less than three times the number of vertices (this is a corollary of Euler's formula [7]).

There still remains the problem of how to organize the q-edges in a leaf's region. We propose to partition the q-edges in a leaf's region into 7 classes, each of which can be ordered by a dictionary. Note that in any given leaf, most of these classes will usually be empty.

The most obvious class of q-edges is the one that meets at a vertex within the leaf's region. This class can be ordered in an angular manner as has been done previously. The remaining q-edges that pass through the leaf's region must enter at one side and leave via another. This yields six classes: NE, NS, NW, EW, SW, and SE, where NE denotes q-edges that intersect both the northern and the eastern boundaries of the leaf's region. Note that the q-edges are undirected edges. For example, the q-edges in class NE (the other 5 classes are handled analogously) are ordered according to whether they lie to the left or to the right of each other when viewing them in an easterly direction from the northern boundary of the leaf's region. Q-edges that coincide with the border of a leaf's region are placed in either NS or EW as is appropriate. Note that a leaf's boundary can coincide with at most one q-edge because if it coincided with two q-edges, then it would have to contain two vertices and thereby violate C1.

Before contemplating the algorithmic aspects of the PM quadtrees, we observe that our proposed quadtrees are relatively compact. For example, the PM_1 quadtree of Figure 6 required 28 quadtree leaf nodes and 31 q-edge nodes (scattered among 24 dictionaries), the PM_2 quadtree of Figure 8 required 16

quadtree leaf nodes and 22 q-edge nodes (scattered among 13 dictionaries), and the PM_3 quadtree of Figure 10 required 7 quadtree leaf nodes and 17 q-edge nodes (scattered among 9 dictionaries). Note that many of the dictionaries consist of single data nodes.

4. ALGORITHMS FOR PM QUADTREES

Now that we have developed the PM quadtree, it is appropriate to examine how it can be used to achieve the three tasks that we specified in Section 1, that is, point location, dynamic line insertion (deletion is discussed in Appendix 1), and map overlay. For the first two tasks, our discussion starts with the PM_1 quadtree, after which we show how the PM_2 and PM_3 quadtrees perform it. For map overlay we focus primarily on the PM_3 quadtree. We first consider point location.

4.1 Point Location

For PM_1 quadtrees (built from C1, C2′, and C3) this problem has three cases which are illustrated by queries with respect to the points X, Y, and Z in Figure 6. The first case is illustrated by the point labeled X. In this case, the point lies in a leaf containing exactly one q-edge. Since region information is linked to each q-edge indicating the regions associated with the q-edge, this reduces to determining the side of the q-edge on which the point lies. Note that the cost of determining the region information associated with a q-edge is also, at most proportional to A1.

The second case is illustrated by the point labeled Y. In this case, the query point lies in a leaf containing a vertex, C in this example. This situation reduces to finding a q-edge in the dictionary that would neighbor a hypothetical q-edge from C and passing through Y. Such a neighboring q-edge must border the region containing Y. Thus, once again our task is reduced to determining on which side of a q-edge a point lies (i.e., Y).

The third case is illustrated by the point labeled Z. In this case, the query point lies in a leaf, say q, containing no q-edges. This means that all the points in the region represented by the leaf q-lie in the same region of the polygonal map. It also means that one of q's brothers must be the root of a subtree that contains a q-edge that borders the region containing Z. In order to find this (not necessarily unique) brother, we visit the brothers in an arbitrary order, say counterclockwise in this example. When considering a brother, one of two cases arises. Either q's brother contains a q-edge (the first case), say b, or it doesn't (the second case). In the first case, the problem reduces to determining the side of q-edge b on which Z lies. This is accomplished by postulating a hypothetical point Z′ in region r that is infinitesimally close to q's region and recursively reapplying the point location procedure to Z′. As an example consider point Z in Figure 6. Since q, the leaf containing Z, is empty, we examine its counterclockwise brother, say r, and postulate a point Z′ that is just across the boundary between q and r. Determining the polygon in which Z′ lies (in this example) is equivalent to determining the polygon in which X lies. In the second case, r contains no q-edges and the algorithm proceeds to examine r's counterclockwise brother.

It should be clear that one of the brothers must contain a q-edge, otherwise the brothers would have been merged to yield a larger node.

The worst-case execution time of point location using a PM_1 quadtree constructed with criteria C1, C2′, and C3 is proportional to the depth of the entire structure—that is, the depth of the quadtree built from C1, C2′, and C3 plus A1 (where A1 is the maximum depth of a dictionary structure in a quadtree leaf node).

Replacement of C3 by C3′, resulting in a PM_2 quadtree, does not lead to significant changes in the point location procedure. The situation arising when q-edges are ordered about a point exterior to their region is handled in the same way as q-edges that are angularly ordered about their point of intersection. It is convenient to store, with each dictionary, the point about which the ordering is being performed, although this can be avoided by sampling two q-edges from the dictionary.

Point location in PM_3 quadtrees is accomplished by finding the closest bordering q-edge with respect to each of the seven classes. The closest q-edge of these seven q-edges borders the region containing the query point. The worst-case cost of point location when using a PM_3 quadtree is proportional to D1 plus A1. This is because the cost of finding the appropriate quadtree node is proportional to D1, and the cost of finding a q-edge that borders the region containing the point from the set of q-edges that are in the node is proportional to A1. The proportionality to A1 is a result of the following. For each of the classes of q-edges, we find the q-edge from that class that is closest to the point. For the class of q-edges that meet at a vertex within the node, this is the same process as used to locate a point within a node of a PM_1 quadtree. For the other classes, it is similar except that instead of relying on angles, we must actually calculate on which side of a line a point lies. Again, for each of these classes the associated worst-case cost is proportional to A1. Now, we must decide which of these closest q-edges (at most 7), forming the set E, is actually part of the border of the region containing the point. Since the only way one of these q-edges could fail to be part of the border is if there was a portion of the border between that q-edge and the point. It is sufficient to determine which q-edge of E is closest to the point in order to know that no other q-edge separates it from the point. This last determination can be done in constant time and thus the entire processing of a node can be done in time proportional to A1.

4.2 Dynamic Line Segment Insertion in PM Quadtrees

Initially, let us assume that we are given a PM_1 quadtree. To insert an edge AB, we insert a q-edge of AB into each quadrant intersected by AB. In some of these quadrants, the insertion of a q-edge of AB would cause a violation of one of the criteria. In that case, the quadrant in question is subdivided and insertion is reattempted. Insertion in PM_2 and PM_3 quadtrees is done in the same manner. The actual algorithms for insertion of line segments as well as deletion for all three types of PM quadtrees are given in Appendix 1.

The above subdivision of a quadrant can cause q-edges of edges that had been previously inserted to be further subdivided. For example, consider Figure 11.

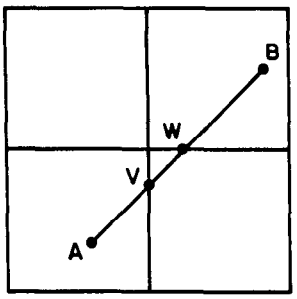
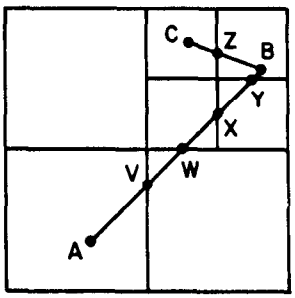


Fig. 11. Result of inserting line segments into the PM_1 quadtree.



First, we insert the edge AB, which entails inserting the q-edges: AV, VW, and WB. Now, suppose that we insert the edge BC. This entails not only inserting the q-edges CZ and ZB, but also the q-edges WX, XY, and YB. Thus, the ultimate cost of inserting an edge into a PM_1 quadtree is often paid for over many insertions as q-edges of the edge are further subdivided to accommodate edges that are being subsequently added.

In order to handle this situation, for our worst-case analysis we do not consider the total cost of inserting a particular edge in a tree. Instead, we consider the ultimate cost of inserting that portion of the map that is currently built. This cost, henceforth known as the *running-sum worst-case cost*, assumes that the map is being built dynamically, that is, information about future edges is not exploited at the time an edge is initially inserted. Note that the running-sum worst-case cost (when summed over the insertions that built the map) is an upperbound on the actual cost of building the map so far. Implicit in the calculation of the running-sum worst-case cost at any instant during the building of a map is the assumption that we know the ultimate depth to which the tree will be expanded. This approach to cost analysis is related to the "amortization" method [2] in that the real difference between the running-sum worst-case cost of the map before and after an insertion is equivalent to the "amortized" cost of that insertion.

The running-sum worst-case map building cost is the product of the cost of inserting a q-edge and the number of q-edges that would have to be inserted. The cost of inserting a q-edge is the depth DMAX of the quadtree (the maximum of

D1, D2′, and D3) plus the depth of the dictionary structure (A1). The calculation of an upperbound for the number of q-edges is slightly more complicated. We define L, the length of the perimeter of a polygonal map, to be the sum of the lengths of all the edges that form the map. Recall that the length of a side of the bounding square is 1. In the following we show that the upperbound on the number of q-edges in the representation of the map is a function of L and the maximum depth, DMAX, of the quadtree structure.

Let us consider the structure of the q-edges that form a single edge. First, we note that for each edge there are at most two q-edges that have the property of being incident at one of the vertices of the graph. Thus the number of such q-edges is proportional to the number of edges in the graph. Since the factor D1 is both bounded from above by DMAX and requires that no edge is less than 2^{-DMAX} units long, we deduce the following upperbound on the number of edges, denoted by E, in a map.

$$E \cdot 2^{-DMAX} \leq L$$

or

$$E \leq L \cdot 2^{DMAX}.$$

Of the remaining P q-edges in the map, all begin and end on the boundary of a square of size 2^{-DMAX} by 2^{-DMAX}. First, we note that while a line segment that passes through a square may be shorter than the width of that square, a line segment that passes through two congruent squares must be at least as long as the width of one of the squares. For each edge, we group together the maximum number of disjoint pairs of contiguous q-edges. The result of this pairing process is that there is at most one unpaired q-edge of the P q-edges processed for each edge. Hence the number of unpaired q-edges is bounded from above by E (the number of edges). Let P′ denote the number of q-edges that remain after the elimination of these unpaired q-edges. These P′ q-edges can be grouped into P′/2 pairs of q-edges that pass through two congruent squares. Note that P′ must be even. This leads to the following upperbound on P′.

$$\frac{P'}{2 \cdot 2^{-DMAX}} \leq L$$

or

$$P' \leq L \cdot 2 \cdot 2^{DMAX}.$$

To summarize, the number of q-edges is equal to the number of q-edges that are incident at a vertex plus the number of q-edges that are left over after the pairing process plus the number of q-edges that participate in the pairing process. For each of these values, we have an upperbound proportional to the length of the perimeter of the map times 2^{DMAX}. Thus the total number of q-edges is also bounded from above by the length of the perimeter of the map times 2^{DMAX}. Recall that the running-sum worst-case map building cost was proportional to the depth of the entire structure (DMAX plus A1) times the number of q-edges inserted, for which we have just derived an upperbound. More formally, an

```
procedure OVERLAY(SUBTREE1, SUBTREE2);
/* Compute the overlay of the quadtrees SUBTREE1 and SUBTREE2. */
begin
  value pointer quadtree SUBTREE1, SUBTREE2;
  pointer quadtree QTD, THE_SUBTREE, TREE_TO_RETURN;
  quadrant X;
  if IS_LEAF(SUBTREE1) and IS_LEAF(SUBTREE2) then
    return(MERGE(SUBTREE1, SUBTREE2)
  else if IS_LEAF(SUBTREE1) or IS_LEAF(SUBTREE2) then
    begin
      QTD ← QUARTER(WHICHEVER_WAS_LEAF(SUBTREE1, SUBTREE2));
      THE_SUBTREE ← WHICHEVER_WAS_NOT_LEAF(SUBTREE1,
                                                SUBTREE2);
      TREE_TO_RETURN ← NEW_NODE( );
      foreach X in {'NW', 'NE', 'SW', 'SE'} do
        SON(TREE_TO_RETURN, X) ← OVERLAY(SON(QTD, X),
                                          SON(THE_SUBTREE,X));
      return(TREE_TO_RETURN);
    end
  else
    begin
      TREE_TO_RETURN ← NEW_NODE( );
      foreach X in {'NW', 'NE', 'SW', 'SE'} do
        SON(TREE_TO_RETURN, X) ← OVERLAY(SON(SUBTREE1, X),
                                          SON(SUBTREE2, X));
      return(TREE_TO_RETURN);
    end;
end;
```

Program I

upperbound on the cost of building a PM quadtree by inserting one edge at a time is given by

$$(E \cdot 3 + L \cdot 2^{DMAX+1}) \cdot (DMAX + A1),$$

which in turn is bounded from above by

$$(5 \cdot L \cdot 2^{DMAX}) \cdot (DMAX + A1).$$

Note that the analysis of dynamic line insertion for PM_2 quadtrees is the same as for PM_1, except that DMAX now denotes the maximum of D1 and D2. Similarly, for the PM_3 quadtree, the analysis need only be modified in that DMAX now corresponds to D1.

4.3 Map Overlay Algorithm for PM Quadtrees

We first consider the computation of map overlay for PM_3 quadtrees. The overlay algorithm can be decomposed into four procedures: OVERLAY, MERGE, CAN_MERGE, and QUARTER. The code for some of them is presented below using a pseudo ALGOL notation in order to provide a maximum amount of information in a minimum amount of space. Procedure OVERLAY (Program I) takes two PM_3 quadtrees as parameters. It traverses the two quadtrees in parallel. When

```
procedure MERGE(LEAF1, LEAF2);
/* Perform the overlay algorithm on the simple case where both quadtrees, LEAF1 and
   LEAF2, are leaf nodes. */
begin
  value pointer quadtree LEAF1, LEAF2;
  pointer quadtree LEAF_TO_RETURN, QT1, QT2;
  quadrant Q;
  dictionary_index X;
  if not CAN_MERGE(LEAF1, LEAF2) then
    begin
      LEAF_TO_RETURN ← NEW_NODE( );
      QT1 ← QUARTER(LEAF1);
      QT2 ← QUARTER(LEAF2);
      foreach Q in {'NW', 'NE', 'SW', 'SE'} do
        SON(LEAF_TO_RETURN, Q) ← MERGE(SON(QT1, Q),SON(QT2, Q));
      return(LEAF_TO_RETURN);
    end
  else
    begin
      LEAF_TO_RETURN ← NEW_NODE( );
      D_VERTEX(LEAF_TO_RETURN) ←
        WHICHEVER_HAD_D_VERTEX(LEAF1, LEAF2);
      foreach X in {'NE', 'NW', 'NS', 'SE', 'SW', 'EW'} do
        D_SIDE(LEAF_TO_RETURN, X) ← D_MERGE(D_SIDE(LEAF1, X),
                                            D_SIDE(LEAF2, X));
    return(LEAF_TO_RETURN);
    end;
end;
```

Program II

one tree is a leaf and the other tree is not, the leaf is split into a node with four sons, each of which are leaf nodes (and correspond to a description of the same region as the original leaf) and the OVERLAY procedure is applied recursively to the corresponding sons. When both quadtrees are leaf nodes, the dictionaries of q-edges in each of them are merged to form a leaf in the output tree. A node is represented as a record with a number of fields. The dictionaries are accessed by the D_VERTEX and D_SIDE fields. D_VERTEX refers to the dictionary associated with the vertex. D_SIDE refers to the remaining dictionaries which are accessed with the aid of dictionary indices {'NE', 'NW', 'NS', 'SE', 'SW', 'EW'}.

Procedure MERGE (Program II) produces the subtree that results from merging two leaf nodes (from a pair of PM_3 quadtrees) depending on whether or not the q-edges involved intersect. Recall that the information about q-edges that is stored in the leaf nodes is ordered with respect to various intercepts (either a vertex or a side of the block). Thus the merger of this information is simply the merger of the corresponding trees. The routine that performs the actual merging is termed D_MERGE and is not given here. The worst-case execution time of MERGE is proportional to the number of nodes merged plus the cost of executing the procedures: CAN_MERGE and QUARTER.

The coding of procedure MERGE uses WHICHEVER_HAD_D_VERTEX, which returns NIL if neither leaf contains a vertex and otherwise returns the dictionary connected to the vertex. Note that the function CAN_MERGE has a

```
Boolean procedure CAN_MERGE(LEAF1, LEAF2);
/* Returns true if and only if the merger of the leaf nodes, LEAF1 and LEAF2, would
   not create any new vertices. Note that in the case that neither leaf node contains a
   vertex, it is possible for one intersection to occur and yet the nodes would still be
   mergible. The counter, N, records the number of known vertices in the pair of nodes.
   If this counter is zero, then LINES_INTERSECT, upon noticing that exactly one
   intersection occurs, has the side effect of incrementing N and updating the
   D_VERTEX field of its last parameter, which is always LEAF1. Of course, if more
   than one intersection occurs, then LINES_INTERSECT will cause CAN_MERGE to
   return false. */
begin
  reference pointer quadtree LEAF1, LEAF2;
  dictionary_index X, Y;
  dictionary THE_VERTEX;
  integer N;
  N ← 0;
  if HAS_VERTEX(LEAF1) and HAS_VERTEX(LEAF2) then
    if SAME_XY_VERTEX(LEAF1, LEAF2) then
      begin
        D_VERTEX(LEAF1)←D_MERGE(D_VERTEX(LEAF1),
                                D_VERTEX(LEAF2));
        D_VERTEX(LEAF2) ← nil;
      end
    else return(false);
  if HAS_VERTEX(LEAF1) or HAS_VERTEX(LEAF2) then
    begin
      N ← 1;
      THE_VERTEX ← D_VERTEX(WHICHEVER_HAD_VERTEX(LEAF1,
                                                 LEAF2);
      foreach X in {'NE', 'NW', 'NS', 'SW', 'SE', 'EW'} do
        if LINES_INTERSECT(THE_VERTEX, D_SIDE(LEAF1, X), N, LEAF1)
          or LINES_INTERSECT(THE_VERTEX, D_SIDE(LEAF2, X), N, LEAF1)
          then return(false);
    end;
  foreach X in {'NE', 'NW', 'NS', 'SW', 'SE', 'EW'} do
    begin
      foreach Y in {'NE', 'NW', 'NS', 'SW', 'SE', 'EW'} do
        begin
          if LINES_INTERSECT(D_SIDE(LEAF1, X), D_SIDE(LEAF2, Y), N,
                             LEAF1)
            then return(false);
        end;
    end;
  return(true);
end;
```

Program III

side effect of removing redundant references to the same vertex (i.e., with the same x and y coordinates). The information in two dictionaries is merged to form a new dictionary by the function D_MERGE.

Procedure CAN_MERGE (Program III) determines whether a pair of leaf nodes of PM_3 quadtrees can be merged. In order to be mergible, the q-edges in the two leaf nodes cannot intersect and if there is a vertex in both of the leaf nodes, then it must have the same x and y coordinate values. Since the checking of intersection (done by the procedure LINES_INTERSECT) can take advantage

of the ordering of the q-edges, the execution time of CAN_MERGE is proportional to the number of q-edges in its leaf parameters.

The final procedure to consider is QUARTER, which takes a leaf as a parameter and returns a subtree containing four leaves that represents the same map. This procedure involves visiting each q-edge in its leaf parameter and determining which parts of it will lie in which sons of the new subtree. Its execution time is proportional to the number of q-edges processed. We don't give its code here. It can be coded easily using the primitives presented in Appendix 1. Note that the code in this section is presented on a more detailed level than the code in Appendix 1.

We now consider an analysis of the OVERLAY algorithm. Let N be the number of q-edges in the PM quadtree built by OVERLAY. Recall that DMAX is an upperbound on the depth of the PM quadtree (not including the depth of the dictionary structures). Although OVERLAY performs a preorder traversal of two subtrees in parallel, its calculation could be performed by a breadth-first traversal of the two trees. This reformulation is used in the following analysis.

First, we note that the cost of executing OVERLAY is proportional to the number of nodes in the input quadtrees (as is the case for the region quadtree intersection and union algorithms) except that the cost of the CAN_MERGE and QUARTER procedures is unbounded with respect to the size of the input tree (whereas the analogous procedures for a region quadtree are bounded). Instead, we find that the cost of performing the CAN_MERGE or QUARTER procedure is proportional to the number of q-edges in its two leaf parameters. We now consider the worst case of the execution time of the OVERLAY algorithm. This occurs when we overlay a pair of PM quadtrees having two corresponding leaf nodes at level k containing vertices which are arbitrarily close to each other. Alternatively, this worst case also results when two new intersection points are created that are arbitrarily close to each other. With respect to the analysis, the significance of two vertices being arbitrarily close to each other is that DMAX becomes arbitrarily larger than k.

Let i be a level between k and DMAX. The execution of OVERLAY on each leaf at level i will result in an invocation of CAN_MERGE and QUARTER (which creates the leaf nodes of level $i + 1$). Hence the cost associated with processing at level i is proportional to the number of q-edges at level i. This cost must be paid at each of the levels between k and DMAX. The number of such levels is bounded from above by DMAX. The number of q-edges at any given level is bounded from above by the number of q-edges in the final result, that is, N. Thus, it follows that the entire OVERLAY algorithm will execute in time proportional to N·DMAX.

Similar algorithms for map overlay can be devised for the PM_1 and PM_2 quadtrees. The above analysis holds for PM_1, PM_2, and PM_3 quadtrees. At first glance, it might appear that the OVERLAY algorithm could be executed just as effectively by repeatedly performing dynamic line insertions from one of the PM quadtrees into the other. However, the analysis for such an approach turns out to be of order N·(DMAX + A1). Our OVERLAY procedure does better than this because q-edges occurring within a given dictionary are processed sequentially instead of randomly.

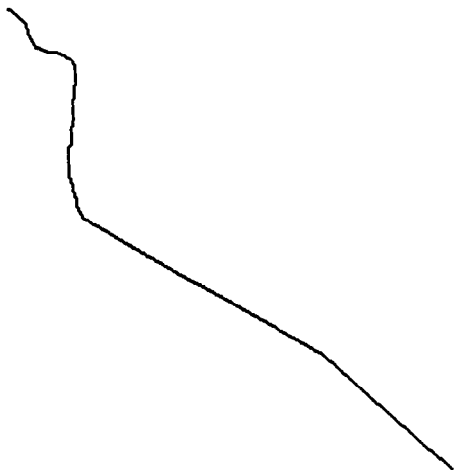


Fig. 12. The powerline map.

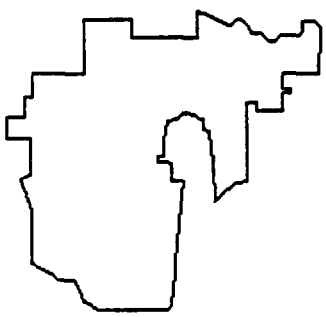


Fig. 13. The cityline map.

5. EMPIRICAL RESULTS

The variants of the PM quadtree discussed in Section 3 were used to encode three maps (Figures 12 through 14) chosen from a cartographic database with which we have been working in our prior experiments with quadtrees [20]. The powerline map (Figure 12) shows the path of a main powerline through a section of the Russian River area in California. The cityline map (Figure 13) indicates the border of the local municipality. The roadline map (Figure 14) is our most complicated map, which details a part of the local roadway network. Table I contains the number of vertices and edges in each of these maps. Note that all of these maps consist of line segments whose vertices rest on a 512 by 512 grid that is offset by half a pixel width from the coordinates of the lower left-hand corners of the quadtree nodes at level 9 (later we will consider the impact of this displacement).

As mentioned in Section 2, neither the MX quadtree nor the edge quadtree is really an appropriate representation for polygonal maps since they only correspond to an approximation (or in the case of the MX quadtree, a digitization) of the map, whereas the variants of the PM quadtree represent the maps exactly.

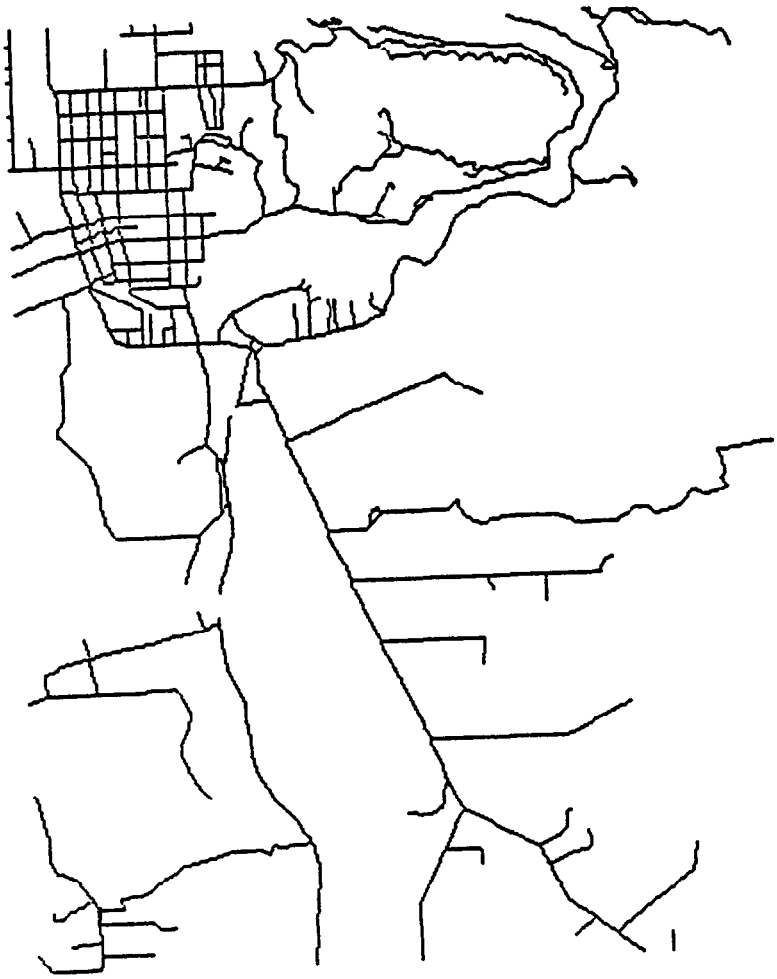


Fig. 14. The roadline map.

Nevertheless, in practice, for the MX quadtree it is natural to consider the approximation that results from representing line segments with the same accuracy as the grid. For the 512 by 512 images that we are considering, this means that the MX quadtree is built by truncating the decomposition at depth 9. Similarly, the edge quadtree is also constructed by truncating the decomposition at depth 9.

Tables II–V summarize the storage requirements of the various quadtree methods of representing the maps of Figures 12–14. As we observed before, the PM_1 quadtree will always be the largest of the PM quadtrees—that is, it will require the most nodes. Therefore, let us consider how it compares with two alternative approaches, the MX and edge quadtrees given in Tables II and III, respectively. Tables IV and V contain the data for the different PM quadtrees. Table V breaks down the leaf count in terms of the different type of nodes as

Table I. Size of the Maps

Map	No. of vertices	No. of edges
Powerline	15	14
Cityline	64	64
Roadline	684	764

Table II. Size of the MX Quadtrees

Map	Depth	Leaves	BLACK nodes	WHITE nodes
Powerline	9	1594	521	1073
Cityline	9	2335	782	1553
Roadline	9	19513	7055	12458

Table III. Size of the Edge Quadtrees

Map	Depth	Leaves	Vertex nodes	Line nodes	WHITE nodes
Powerline	9	211	15	68	128
Cityline	9	730	64	219	447
Roadline	9	6658	684	2431	3543

Table IV. Size of the PM_1, PM_2, and PM_3 Quadtrees

	Depth			Leaves			Q-edges		
Map	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
Powerline	7	7	7	61	61	61	38	38	38
Cityline	9	8	8	214	208	187	178	176	168
Roadline	13	9	9	2125	1960	1714	2144	2096	1976

Table V. Breakdown of Information in Table IV by Node Type

	Vertex nodes (average q-edges)			Line nodes (average q-edges)			WHITE nodes		
Map	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
Powerline	15 (1.9)	15 (1.9)	15 (1.9)	10	10 (1.0)	10 (1.0)	36	36	36
Cityline	64 (2.0)	64 (2.0)	64 (2.1)	50	47 (1.0)	33 (1.0)	100	97	90
Roadline	684 (2.2)	684 (2.2)	684 (2.3)	618	515 (1.1)	360 (1.1)	823	761	670

well as gives the average number of q-edges for each node type (in parentheses) where it is relevant.

The MX quadtree (see Table II) has the worst performance. In all of our examples the MX quadtree is larger than the PM_1 quadtree (see Table IV) by at least a factor of 9. More generally, we would expect the size of the MX quadtree for a polygonal map to be roughly as large as the product of the average line length and the number of nodes in the corresponding PM_1 quadtree. This can be seen by the following chain of arguments. First, for "typical data," the number of nodes in a PM_1 quadtree of a polygonal map is proportional to the number of vertices in the polygonal map since, typically, the vertex nodes are the lowest

nodes in the PM_1 quadtree (although there are rare exceptions as illustrated in Figure 7). In the analysis of quadtrees, a good rule of thumb is that the deepest frequently occurring node type will dominate the size measurement. Second, Hunter and Steiglitz [9, 10] have shown that the number of nodes in an MX quadtree of a polygonal map is proportional to the perimeter of the polygon. Third, we know that polygonal maps are planar maps which means that the number of line segments in each map is proportional to the number of vertices in the map. Combining these three arguments with the fact that the perimeter of a map is equal to the product of the number of line segments and the average length of a line segment leads to the desired result—that is, typically, the number of nodes in the MX quadtree is of the order of the product of the number of vertices in the PM_1 quadtree and the average line length (measured in pixels).

Now, let us compare the edge quadtree with the PM_1 quadtree. The edge quadtree (see Table III) can be seen to be a definite improvement over the MX quadtree. Considering the trivial (but often typical) maps like powerline and cityline, we see that the edge quadtree is about three times as large as the PM_1 quadtree. This can be explained by observing that the average depth of a vertex node in the PM_1 quadtree for each of these two maps was between 6 and 7 (not shown in our tables) whereas the corresponding edge quadtree must represent all of the vertex nodes at depth 9.

The roadline map, which is the most complex map that we have examined to date, has an edge quadtree that is also about three times the size of its PM_1 quadtree. This might at first appear surprising since we observe that the maximum depth of this quadtree is considerably greater than that required by the digitization grid. In this case the digitization grid requires a depth of 9 while the PM_1 quadtree requires some nodes to be at a depth of 13. Although for this map, the maximum depth of the PM_1 quadtree is greater (i.e., 13) than that of the edge quadtree (i.e., 9), we can explain the difference in the number of nodes in the two trees by examining the distribution of nodes by depth (see Table VI). In essence, the average depth of a vertex node is again between 6 and 7 for the PM_1 quadtree while it is 9 for the edge quadtree. The reduction in the average depth of a vertex node in the PM_1 quadtree has a direct effect on the total number of nodes because the decomposition of a line is identical in the edge and PM_1 quadtrees once the line segment has exited the region of the vertex nodes representing its endpoints.

The above discussion leads us to conclude that the PM_1 quadtree is an improvement over earlier approaches to handling real data. We have seen that the PM_1 quadtree has the desirable property of reducing the average depth at which the dominant node type is located. The PM_2 and PM_3 quadtrees are attempts to further reduce the maximum depth of nodes in a PM_1 quadtree. The PM_2 quadtree has the effect of reducing the maximum depth (see Table VI) by virtue of a more compact treatment of the case when close edges that radiate from the same vertex lie in a different node from the vertex. When comparing the data of the PM_1 quadtree columns with the data of the PM_2 quadtree columns of Table IV, we observe no change in the powerline map since it is composed of only obtuse angles. The cityline map has a few acute angles creating situations where line nodes can be formed containing more than one q-edge, thus causing some of the line nodes to be closer to the root and resulting in a 3 percent

Table VI. Distribution of Node Types by Depth for PM Quadtrees in the Roadline Map

	Vertex nodes			Line nodes			WHITE nodes		
Depth	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	4	4	4
3	2	2	2	0	0	0	8	8	8
4	10	10	12	4	6	7	38	38	38
5	75	75	82	34	35	29	109	105	101
6	180	180	192	132	127	120	218	212	199
7	224	224	232	204	198	139	237	222	195
8	158	158	135	156	126	59	162	142	104
9	35	35	29	48	23	6	40	30	21
10	0			13			2		
11	0			14			3		
12	0			10			1		
13	0			3			1		

reduction in size. The more complicated roadline map presents more such situations resulting in an 8 percent reduction in size. We note that the PM_2 reduction only affects the depth of the line nodes. Recall that nodes containing a vertex are treated in the same manner in both the PM_1 and PM_2 quadtrees. This observation is reinforced by noting that the vertex node columns in Table VI, which show the distribution of node type by depth, are identical.

Comparing the PM_3 quadtree with the PM_1 and PM_2 quadtrees also shows no change in the number of nodes when used on the powerline map. This is because the powerline map contains no edges that pass closely to vertices other than their endpoints. This situation occurs a bit more frequently in the cityline map resulting in a 12 percent reduction in size. Note that the existence of such situations implies that vertex nodes will be slightly closer to the root in the PM_3 quadtree than in the PM_1 and PM_2 quadtrees. For the roadline map, the use of the PM_3 quadtree instead of a PM_1 quadtree leads to a 19 percent reduction in size. This is due to the tendency for vertex nodes to occur closer to the root in the PM_3 quadtree than in the PM_1 quadtree and can be seen by examining Table VI.

From the above we see that although the differences among the different PM quadtrees can be drastic in principle, for typical cartographic data, the difference in the number of nodes in the various PM quadtrees for a particular map is less pronounced. Thus, for cartographic data, the choice among the different PM quadtrees is dictated more by the problems of implementation rather than by the need to conserve space. However, it should be noted that cartographic data is rather special in that it generally consists of sequences of short line segments meeting at obtuse angles. Since the lengths of the line segments are often shorter than the distance between the features that the line segments are representing, this yields data that tends to bring out the best in each of the types of PM quadtrees with the result that there is little difference between them. For data that is not this simple, the benefits of the PM_2 and PM_3 quadtrees over the PM_1 quadtree should be more pronounced.

Table VII. The Effect of Small Shifts on Different Representations of the Roadline Map

	Number of leaves resulting from different shifts					
Quadtree type	0.5	0.625	0.75	0.0	0.125	0.25
MX	19513	19720	19627	20554	19597	19618
Edge	6658	6709	6757	8611	6760	6727
PM_1	2125	2179	2218	2698	2275	2203
PM_2	1960	1984	1981	2608	1999	1993
PM_3	1714	1714	1714	2434	1714	1714

Aside from the consideration of the number of leaves in the various quadtree implementations, there are two further aspects of storage to be examined: 1) the number of q-edges in the various quadtree nodes and 2) the sensitivity of the PM quadtree representations to slight shifts in the placement of the data.

In Table IV we find that a reduction in the number of q-edges closely parallels the reduction of the number of quadtree leaf nodes across the different PM quadtree implementations. Table V tabulates the average number of q-edges per node of a particular node type. This is placed in parentheses immediately after the count of the number of leaf nodes of that node type. No averages are given for WHITE nodes and PM_1 quadtree line nodes as by definition either they have zero or one q-edge, respectively. Investigation of the average number of q-edges per line node shows that it is rare for there to be more than one q-edge per line node. In the case of vertex nodes, we find that the number of q-edges per node is consistently around 2 although it does seem to increase slowly with map complexity. These values tend to indicate that a linked list is usually sufficient to organize the q-edges at a given node instead of the 2–3 tree as advocated in Section 3.2. In the PM quadtree implementation of our three test maps, we found at most one node in a given map that had as many as 5 q-edges. Thus not only is the average low, but there does not seem to be much variance from the average value either.

In Section 2, we stated that one of the motivations for the development of the PM quadtree data structure is that its size is relatively invariant to shifting and rotation. Table VII summarizes the results of some experiments on the effect of minor shifts in positioning the vertices of the roadline map. The first column, labeled 0.5, shows the data used to generate Tables II–VI. Recall that to obtain these tables we shifted our original data by adding 0.5 to what were originally integer coordinates on a 512 by 512 grid. The column labeled 0.0 indicates no change in positioning the vertices of the original data and shows significantly higher node counts than the other shifts. This is not surprising since, as stated in Section 2.2, when a vertex lies on the border of a quadtree node it is inserted in each of the nodes whose border it touches. This can cause further node splits if some of the quadtree nodes into which it is inserted already have a vertex in them. However, if the vertices that lie on the borders of quadtree nodes are shifted slightly, then they no longer will share a quadtree node and thus no further decomposition will be required. Once the placement of vertices on the borders of quadtree nodes has been avoided (e.g., by using small shifts), there still remains the secondary effect that vertices close to the border of a quadtree node tend to result in a very small separation between the q-edges in the

neighboring quadtree node. This has the greatest effect on the number of nodes in the PM_1 quadtree, while it has no effect on the number of nodes in the PM_3 quadtree. Like the PM_3 quadtree, the PM_2 quadtree is not affected by q-edges whose separation is small because they result from a vertex being near a quadtree boundary. However, this is not shown so clearly by the entries in Table VII for the PM_2 quadtree since the PM_2 quadtree is susceptible to the digitization effects that result from the process of determining whether or not a line falls within a particular square region. The only digitization effects that can alter the number of leaf nodes in the PM_3 quadtree are those resulting from the process of determining whether or not a vertex lies within a particular square.

6. CONCLUDING REMARKS

We have taken an iterative approach to the development of a quadtree-like data structure for storing polygonal maps. We started with the PR quadtree and developed the PM quadtrees. The final formulation, PM_3, uses the same decomposition rule as the PR quadtree but stores considerably more information in the terminal nodes. The PM quadtree enables storing polygonal maps without information loss. Since isolated vertices pose no problems, the PM quadtree can be used to represent points, lines, and regions. We have shown that point location using the PM quadtree can be performed in time proportional to the depth of the structure, have given an upperbound on the worst-case cost of insertion of a portion of a map dynamically, and have shown how to overlay two polygonal maps that are represented by PM quadtrees. Empirical results on the storage requirements of PM quadtrees were also presented and analyzed showing that the theoretical analysis was overly pessimistic for typical data.

Some future work includes the development and analysis of algorithms for other operations, that is, shift and rotation. Currently, the best known algorithm for shifting or rotating a PM quadtree is to extract each line segment from the source quadtree, transform it, and then reinsert it into the target quadtree. However, it would appear that something better could be achieved since spatial order is preserved under shift and rotation transformations. The simple heuristic of reinserting all the line segments extracted from the same node into the target quadtree together (in a manner analogous to the map-overlay algorithm) may prove useful here.

APPENDIX 1. Insertion and Deletion Routines for PM Quadtrees

Since the PM quadtree is used to implement polygonal maps, its basic entities are vertices and edges. Each vertex is represented as a record of type *point* which has two fields called XCOORD and YCOORD that correspond to the x and y coordinates, respectively, of the point. They can be of type real or integer depending on implementation considerations such as floating point precision. An edge is implemented as a record of type *line* with four fields, P1, P2, LEFT, and RIGHT. P1 and P2 contain pointers to the records containing the edge's vertices. LEFT and RIGHT are pointers to structures that identify the regions which are on the two sides of the edge. We shall use the convention that LEFT and RIGHT are with respect to a view of the edge that treats the vertex closest to the origin as the start of the edge. For example, in Figure 2 the LEFT and RIGHT fields

are marked as being associated with regions 1 and 2, respectively. The algorithms that we give here ignore these two fields. For a discussion of the maintenance of the information that they store, see Appendix 3.

Each node in a PM quadtree is a collection of q-edges which is organized according to the variant being implemented (i.e., PM_1, PM_2, or PM_3) and is represented as a record of type *node* containing seven fields. The first four fields contain pointers to the node's four sons corresponding to the directions (i.e., quadrants) NW, NE, SW, and SE. If P is a pointer to a node and I is a quadrant, then these fields are referenced as SON(P, I). The fifth field, NODETYPE, indicates whether the node is a terminal node (LEAF) or a nonterminal (GRAY) node. The SQUARE field is a pointer to a record of type *square* which indicates the size of the block corresponding to the node. It is defined for both LEAF and GRAY nodes. It has two fields, CENTER and LEN. CENTER points to a record of type point which contains the x and y coordinates of the center of the square. LEN contains the length of a side of the square which is the block corresponding to the node in the PM quadtree. DICTIONARY is the last field and it is a pointer to a data structure that represents the set of q-edges that are associated with the node. Initially, the universe is empty and consists of no edges or vertices. It is represented by a tree of one LEAF node whose DICTIONARY field points to the empty set.

In the implementation given here the set of q-edges for each LEAF node is a linked list whose elements are records of type *edgelist* containing two fields DATA and NEXT. DATA points to a record of type line corresponding to the edge of which the q-edge is a member. NEXT points to the record corresponding to the next q-edge in the list of q-edges. Although the set of q-edges is implemented as a list here, it really should be implemented by a data structure that supports the efficient execution of the delete, insert, set union, and set difference operations (e.g., 2–3 trees). However, a linked list is usually sufficient since in our empirical tests, described in Section 5, the list rarely had as many as five items in it. Depending on the type of PM quadtree that is being used the set of q-edges could be further decomposed into subsets. For example, in the case of a PM_3 quadtree we would want to have seven subsets corresponding to the vertex and the six combinations of sides. These subsets have been discussed in Section 4.3 in the presentation of the map overlay algorithm and are referred to as D_VERTEX and D_SIDE, respectively. The set of q-edges corresponding to a GRAY node is said to be empty. Note that all of the q-edges comprising a given edge point to the same line record.

An edge is inserted into a PM quadtree by traversing the tree in preorder and successively clipping it (using CLIP_LINES) against the blocks corresponding to the nodes. Clipping is important because it enables us to avoid looking at areas where the edge cannot be inserted. This process is controlled by procedure PMINSERT which actually inserts a list of edges. If the edge can be inserted into the node, say P, then PMINSERT does so and exits. Otherwise, a list, say L, is formed containing the edge and any q-edges already present in the node, P is split, and PMINSERT is recursively invoked to attempt to insert the elements of L in the four sons of P. PMINSERT uses PM_CHECK (i.e., PM1_CHECK, PM2_CHECK, or PM3_CHECK) to determine if the criteria of the appropriate PM quadtree are satisfied. Isolated vertices pose no problems and are handled

by PM_CHECK. The implementation given below assumes that whenever an edge or an isolated vertex is inserted into the PM quadtree it is not already there or does not intersect an existing edge. However, an endpoint of the edge may intersect an existing vertex as long as it is not an isolated vertex. Procedure CLIP_SQUARE is a predicate that indicates if an edge crosses a square. Similarly, procedure PT_IN_SQUARE is a predicate that indicates if a vertex lies in a square. They are responsible for enforcing the conventions with respect to vertices and edges that lie on the boundaries of blocks. Their code is not given here. Equality between records corresponding to vertices is tested by use of the '=' symbol which requires that its two operands be of the same type (i.e., pointers to records of type point).

An edge is deleted from a PM quadtree by using a process whose control structure is identical to that used in the insertion of an edge. Again, the tree is traversed in preorder and the edge is successively clipped (using CLIP_LINES) against the blocks corresponding to the nodes. This process is controlled by procedure PMDELETE which actually deletes a list of edges. At each LEAF node, the DICTIONARY field is updated to show the elimination of the edge (or edges). Once all four sons of a GRAY node have been processed, an attempt is made to merge the four sons by use of procedures POSSIBLE_PM_MERGE (i.e., POSSIBLE_PM1_MERGE or POSSIBLE_PM23_MERGE) and TRYTOMERGE_PM (i.e., TRYTOMERGE_PM1 or TRYTOMERGE_PM23) to check if the criteria of the appropriate PM quadtree are satisfied. These procedures make use of PM_CHECK.

```
recursive procedure PMINSERT(P, R);
/* Insert the list of edges pointed at by P in the PM quadtree rooted at R. The call to
   procedure PM_CHECK is generic. It is replaced by its analog for the PM1, PM2, and
   PM3 quadtrees. */
begin
  value pointer edgelist P;
  value pointer node R;
  pointer edgelist L;
  quadrant I;
  L ← CLIP_LINES(P, SQUARE(R));
  if empty(L) then return;  /* No new edges belong in the quadrant */
  if LEAF(R) then  /* A terminal node */
    begin
      L ← MERGE_LISTS(L, DICTIONARY(R));
      if PM_CHECK(L, SQUARE(R)) then
        begin
          DICTIONARY(R) ← L;
          return;
        end
      else SPLIT_PM_NODE(R);
    end;
  for I in {'NW', 'NE', 'SW', 'SE'} do PMINSERT(L, SON(R, I));
end;

recursive edgelist procedure CLIP_LINES(L, R);
/* Collect all of the edges in the list of edges pointed at by P that intersect the square
   pointed at by R. ADD_TO_LIST(X, S) adds element X to the list S and returns a
   pointer to the resulting list. */
```

```
begin
  value pointer edgelist L;
  value pointer square R;
  return(if empty(L) then NIL
         else if CLIP_SQUARE(DATA(L), R) then
           ADD_TO_LIST(DATA(L), CLIP_LINES(NEXT(L), R))
         else CLIP_LINES(NEXT(L), R));
end;

Boolean procedure PM1_CHECK(L, S);
/* Determine if the square pointed at by S and the list of edges pointed at by L form a
   legal PM1 quadtree node. ONE_ELEMENT(L) is a predicate that indicates if L
   contains just one element. */
begin
  value pointer edgelist L;
  value pointer square S;
  return(if P1(DATA(L)) = P2(DATA(L)) then
           ONE_ELEMENT(L)  /* Isolated vertex */
         else if ONE_ELEMENT(L) then
         /* Both vertices can lie outside the square */
           not(PT_IN_SQUARE(P1(DATA(L)), S) and
               PT_IN_SQUARE(P2(DATA(L)), S))
         else if PT_IN_SQUARE(P1(DATA(L)), S) and
                 PT_IN_SQUARE(P2(DATA(L)), S) then false
         else if PT_IN_SQUARE(P1(DATA(L)), S), then
           SHARE_PM1_VERTEX(P1(DATA(L)), NEXT(L), S)
         else if PT_IN_SQUARE(P2(DATA(L)), S) then
           SHARE_PM1_VERTEX(P2(DATA(L)), NEXT(L), S)
         else false);
end;

recursive Boolean procedure SHARE_PM1_VERTEX(P, L, S);
/* The vertex pointed at by P is in the square pointed at by S. Determine if all the edges
   in the list of edges pointed at by L share P and do not have their other vertex within
   S. */
begin
  value pointer point P;
  value pointer edgelist L;
  value pointer square S;
  return(if empty(L) then true
         else if P = P1(DATA(L)) then
           not(PT_IN_SQUARE(P2(DATA(L)), S)) and
           SHARE_PM1_VERTEX(P, NEXT(L), S)
         else if P = P2(DATA(L)) then
           not(PT_IN_SQUARE(P1(DATA(L)), S)) and
           SHARE_PM1_VERTEX(P, NEXT(L), S)
         else false);
end;

Boolean procedure PM2_CHECK(L, S);
/* Determine if the square pointed at by S and the list of edges pointed at by L form a
   legal PM2 quadtree node. SHARE_PM2_VERTEX is invoked with L instead of
   NEXT(L) because the edge might have one vertex in the square and share the other
   vertex with the list, which violates the PM2 quadtree criteria. */
```

```
begin
  value pointer edgelist L;
  value pointer square S;
  return(if P1(DATA(L)) = P2(DATA(L)) then
           ONE_ELEMENT(L)   /* Isolated vertex */
         else if PT_IN_SQUARE(P1(DATA(L)), S) and
           PT_IN_SQUARE(P2(DATA(L)), S) then false
         else if SHARE_PM2_VERTEX(P1(DATA(L)), L, S) or
                 SHARE_PM2_VERTEX(P2(DATA(L)), L, S) then true
         else false);
end;

recursive Boolean procedure SHARE_PM2_VERTEX(P, L, S);
/* The vertex pointed at by P is the shared vertex in a PM2 quadtree. It can be inside or
   outside of the square pointed at by S. Determine if all the edges in the list of edges
   pointed at by L share P and do not have their other vertex within S. */
begin
  value pointer point P;
  value pointer edgelist L;
  value pointer square S;
  return(if P = P1(DATA(L)) then
           not(PT_IN_SQUARE(P2(DATA(L)), S)) and
           SHARE_PM2_VERTEX(P, NEXT(L), S)
         else if P = P2(DATA(L)) then
           not(PT_IN_SQUARE(P1(DATA(L)), S)) and
           SHARE_PM2_VERTEX(P, NEXT(L), S)
         else false);
end;

recursive Boolean procedure PM3_CHECK(L, S);
/* Determine if the square pointed at by S and the list of edges pointed at by L form a
   legal PM3 quadtree node. In order to allow an isolated vertex to coexist in a leaf node
   along with edges that do not intersect it, INF is used to represent a fictitious point at
   (∞, ∞) and serves as the shared vertex in the call to SHARE_PM3_VERTEX. */
begin
  value pointer edgelist L;
  value pointer square S;
  return(if empty(L) then true
         else if P1(DATA(L)) = P2(DATA(L)) then   /* Isolated vertex */
           SHARE_PM3_VERTEX(INF, NEXT(L), S)
         else if PT_IN_SQUARE(P1(DATA(L)), S) and
                 PT_IN_SQUARE(P2(DATA(L)), S) then false
         else if PT_IN_SQUARE(P1(DATA(L)), S) then
           SHARE_PM3_VERTEX(P1(DATA(L)), NEXT(L), S)
         else if PT_IN_SQUARE(P2(DATA(L)), S) then
           SHARE_PM3_VERTEX(P2(DATA(L)), NEXT(L), S)
         else PM3_CHECK(NEXT(L), S));
end;

recursive Boolean procedure SHARE_PM3_VERTEX(P, L, S);
/* The vertex pointed at by P is the shared vertex in a PM3 quadtree. It is inside the
   square pointed at by S. Determine if all the edges in the list of edges pointed at by L
   either share P and do not have their other vertex within S, or do not have either of
   their vertices in S. */
```

```
begin
  value pointer point P;
  value pointer edgelist L;
  value pointer square S;
  return (if empty(L) then true
          else if P = P1(DATA(L)) then
            not (PT_IN_SQUARE(P2(DATA(L)), S)) and
            SHARE_PM3_VERTEX(P, NEXT(L), S)
          else if P = P2(DATA(L)) then
            not(PT_IN_SQUARE(P1(DATA(L)), S)) and
            SHARE_PM3_VERTEX(P, NEXT(L), S)
          else not(PT_IN_SQUARE(P1(DATA(L)), S)) and
            not(PT_IN_SQUARE(P2(DATA(L)), S)) and
            SHARE_PM3_VERTEX(P, NEXT(L), S));
end;

procedure SPLIT_PM_NODE(P);
/* Add four sons to the node pointed at by P and change P to be of type GRAY. */
begin
  value pointer node P;
  quadrant I, J;
  pointer node Q;
  pointer square S;
  /* XF and YF contain multiplicative factors to aid in the location of the centers of the
     quadrant sons while descending the tree */
  preload real array XF['NW', 'NE', 'SW', 'SE'] with −0.25, 0.25, −0.25, 0.25;
  preload real array YF['NW', 'NE', 'SW', 'SE'] with 0.25, 0.25, −0.25, −0.25;
  for I in {'NW', 'NE', 'SW', 'SE'} do
  begin
    Q ← create(node);
    SON(P, I) ← Q;
    for J in {'NW', 'NE', 'SW', 'SE'} do SON(Q, J) ← NIL;
    NODETYPE(Q) ← 'LEAF';
    S ← create(square);
    SQUARE(Q) ← S;
    CENTER(S) ← create(point);
    XCOORD(CENTER(S)) ← XCOORD(CENTER(SQUARE(P))) +
                        XF[I]*LEN(SQUARE(P));
    YCOORD(CENTER(S)) ← YCOORD(CENTER(SQUARE(P))) +
                        YF[I]*LEN(SQUARE(P));
    LEN(S) ← 0.5*LEN(SQUARE(P));
    DICTIONARY(Q) ← NIL;
  end;
  DICTIONARY(P) ← NIL;
  NODETYPE(P) ← 'GRAY';
end;

recursive procedure PMDELETE(P, R);
/* Delete the list of edges pointed at by P from the PM quadtree rooted at R. The calls
   to procedures POSSIBLE_PM_MERGE and TRYTOMERGE_PM are generic. They
   are replaced by their corresponding analogs for the PM1, PM2, and PM3 quadtrees. */
begin
  value pointer edgelist P;
  value pointer node R;
  pointer edgelist L;
  quadrant I;
  L ← CLIP_LINES(P, SQUARE(R));
  if empty(L) then return;  /* None of the edges are in the quadrant */
  if GRAY(R) then
```

```
    begin
      for I in {'NW', 'NE', 'SW', 'SE'} do PMDELETE(L, SON(R, I));
      if POSSIBLE_PM_MERGE(R) then
        begin
          L ← NIL;
          if TRYTOMERGE_PM(R, R, L) then
            begin  /* Merge the sons of the GRAY node */
              RETURN_TREE_TO_AVAIL(R);
              DICTIONARY(R) ← L;
              NODETYPE(R) ← 'LEAF';
            end;
        end;
    end
  else DICTIONARY(R) ← SET_DIFFERENCE(DICTIONARY(R), L);
end;

Boolean procedure POSSIBLE_PM1_MERGE(P);
/* Determine if the subtrees of the four sons of the PM1 quadtree node pointed at by P
   should be further examined to see if a merger is possible. Such a merger is only feasible
   if at least one of the four sons of P is a LEAF. */
begin
  value pointer node P;
  return(LEAF(SON(P, 'NW')) or LEAF(SON(P, 'NE')) or
         LEAF(SON(P, 'SW')) or LEAF(SON(P, 'SE')));
end;

Boolean procedure POSSIBLE_PM23_MERGE(P);
/* Determine if an attempt should be made to merge the four sons of the PM2 or PM3
   quadtree. Such a merger is only feasible if all four sons of a GRAY node are LEAF
   nodes. */
begin
  value pointer node P;
  return(LEAF(SON(P, 'NW')) and LEAF(SON(P, 'NE')) and
         LEAF(SON(P, 'SW')) and LEAF(SON(P, 'SE')));
end;

recursive Boolean procedure TRYTOMERGE_PM1(P, R, L);
/* Determine if the four sons of the PM1 quadtree rooted at node P can be merged. Notice
   that the check for the satisfaction of the PM1 decomposition criteria is with respect to
   the square associated with the original GRAY node, rooted at R, whose subtrees are
   being explored. Variable L is used to collect all of the edges that are present in the
   subtrees. */
begin
  value pointer node P, R;
  reference pointer edgelist L;
  if LEAF(P) then
    begin
      L ← SET_UNION(L, DICTIONARY(P));
      return(true);
    end
  else return(TRYTOMERGE_PM1(SON(P, 'NW'), R, L) and
              TRYTOMERGE_PM1(SON(P, 'NE'), R, L) and
              TRYTOMERGE_PM1(SON(P, 'SW'), R, L) and
              TRYTOMERGE_PM1(SON(P, 'SE'), R, L) and
              PM1_CHECK(L, SQUARE(R)));
end;
```

```
Boolean procedure TRYTOMERGE_PM23(P, R, L);
/* Determine if the four sons of the PM2 or PM3 quadtree rooted at node P can be merged.
   Variable L is used to collect all of the edges that are present in the subtrees. Note that
   there is no need for parameter R, and the procedure is not recursive. The call to PM_
   CHECK is replaced by PM2_CHECK or PM3_CHECK as is appropriate. */
begin
  value pointer node P, R;
  reference pointer edgelist L;
  quadrant I;
  for I in {'NW', 'NE', 'SW', 'SE'} do
    L ← SET_UNION(L, DICTIONARY(SON(P, I)));
  return(PM_CHECK(L, SQUARE(P)));
end;

recursive procedure RETURN_TREE_TO_AVAIL(P);
/* Return the PM quadtree rooted at P to the free storage list. This process is recursive
   in the case of a PM1 quadtree. */
begin
  value pointer node P;
  quadrant I;
  if LEAF(P) then return
  else
    begin
      for I in {'NW', 'NE', 'SW', 'SE'} do
        begin
          RETURN_TREE_TO_AVAIL(SON(P, I));
          returntoavail(SON(P, I));
          SON(P, I) ← NIL;
        end;
    end;
end;
```

APPENDIX 2. Comparison with Nonquadtree Approaches

The problem of constructing a data structure allowing such operations as point location has also been addressed by investigators of geometric complexity [4]. Their work has concentrated on providing a structure with optimal worst-case analysis. One version of an optimal structure that has been reported is the *K-structure* of Kirkpatrick [11].

The K-structure is a hierarchical structure based on triangulation rather than a regular decomposition. The notion of hierarchy in the K-structure is radically different from that of a quadtree, in that instead of replacing a group of triangles by a single triangle at the next higher level, a group of triangles is replaced by a smaller group of triangles. The triangles are first partitioned into groups that share a common vertex. The shared vertex is then eliminated and a new smaller triangulation is then constructed. The size of a K-structure for a polygonal map of v vertices (possibly containing holes) is $O(v)$ although it has a rather high constant of proportionality. Its construction has a worst-case execution time of $O(v)$ for a triangular subdivision and $O(v \cdot \log v)$ for a general map. The latter is dominated by the cost of triangulating the original polygonal map [8].

The point location problem for the K-structure can be solved in $O(\log_2 v)$ time. There is no obvious algorithm that performs overlay using the K-structure. However, since the triangulation forms a convex map, the planar sweep algorithm of Nievergelt and Preparata [14] can be used to perform overlay of two polygonal

maps in $O(v \log_2 v + s \log_2 v)$ time where s denotes the number of intersections. From the results of Nievergelt and Preparata's algorithm, a new K-structure can be built in $O(v)$ time. Thus the total time is $O(v \log_2 v + s \log_2 v)$. The dynamic insertion of a line segment into a K-structure has a worst case of $O(v)$ time.

It should be noted that the optimal worst-case behavior (on data not necessarily fitting the model) of the K-structure for the task of point location is also shared by the *layered dag* of Edelsbrunner, Guibas, and Stolfi [5]. The layered dag is a modified binary tree which stores a y-monotone subdivision of a polygonal map. A *y-monotone subdivision of a polygonal map* is created by partitioning the regions of a map until no vertical line intersects a region's boundary more than twice. Note that the resulting map need not be convex; however, the asymptotic worst-case analysis of the layered dag is identical to that of the K-structure.

Any triangulation can be used as the basis of a y-monotone subdivision, but, usually, the creation of a y-monotone subdivision requires considerably fewer line segments to be inserted than the creation of a triangulation of the same map. However, like triangulation, the creation of a monotone subdivision is performed by a planar sweep algorithm in $O(v \log_2 v)$ time. The layered dag can be built from a y-monotone subdivision in $O(v)$ time and $O(v)$ space. However, as with the K-structure, there is no simple way to perform the overlay algorithm that takes advantage of this structure; so again, we resort to the overlay algorithm of Nievergelt and Preparata [14]. Although the map created by the y-monotone subdivision creation process is not convex, as observed by a referee, one can perform overlay in $O(v \log_2 v + s \log_2 v)$ time. More importantly, it is difficult to perform dynamic update on the layered dag structure.

In comparing the PM quadtree to the K-structure and the layered dag, it is necessary to establish a model of expected data, because the DMAX factor of the quadtree analysis can grow arbitrarily large without changing the v factor of the K-structure and layered dag analyses. Since $\log_4 v$ is only a lower bound on DMAX, the K-structure and the layered dag clearly have a better worst-case performance than the PM quadtree. However, in practice, data seldom exploit this worst-case behavior.

Instead, we observe that cartographic polygonal maps result from a piecewise linear approximation process. This means that the map will tend to contain only long chains of short line segments. Thus the number of q-edges per line segment is usually 2 as is the degree of a vertex. If the cartographic data fit this model exactly, then the edge quadtree discussed in Section 2.3 would be adequate. The PM quadtree is different from those variants in that while it is most efficient for data that closely approximates the model, there is a smooth degradation of its performance as the data departs from this model. Furthermore, the three PM quadtrees presented (PM_1, PM_2, and PM_3) degrade at successively slower rates.

For data that closely fit this model, the PM quadtree has the following characteristics. Point location is performed in $O(\log_2 v)$ time. This follows from the expectation that half the quadtree leaves are empty and the other half usually contain at most two q-edges. Two PM quadtrees can be overlayed in $O(v)$ time. Dynamic insertion of a line segment can be performed in $O(\log_2 v)$ time. This analysis is essentially that of the edge quadtree under ideal data. Note that the expected performance of the PM quadtree for dynamic line insertion and map overlay is better than that expected for the K-structure. For the problem of point

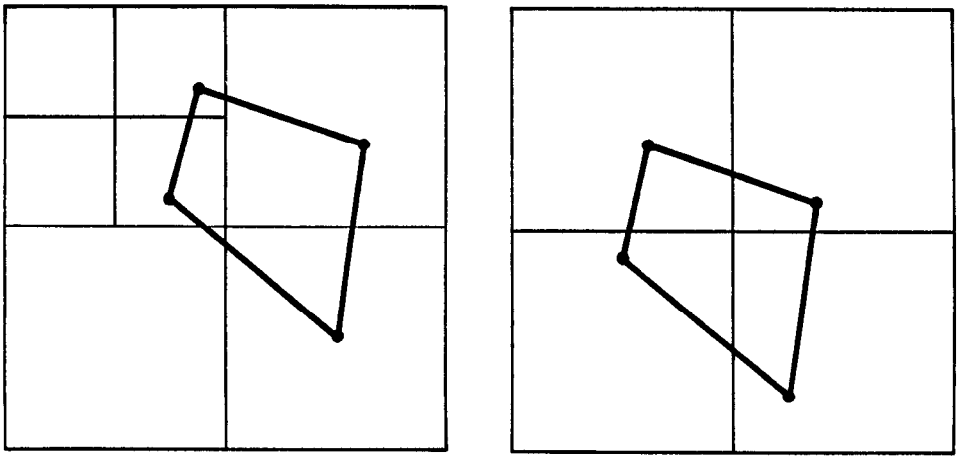


Fig. 15. Example demonstrating the sensitivity of the PR quadtree to shifts.

location the comparison depends on a further investigation of the constant factors involved in the analysis. Although the impact of the model on the triangulation process that underlies the K-structure is difficult to discern, the short line segments will tend to cause the creation of long narrow triangles. This would increase the s factor in the execution time of the overlay algorithm when using the K-structure. Note, also, that applying our model of cartographic data to the layered dag does not change the analysis of the layered dag. Thus for such data, the PM quadtree bears the same relationship to the layered dag as it does to the K-structure.

One difference between the above methods and the quadtree approach is that the size of a quadtree is inherently sensitive to the location of the map within the decomposition grid (and hence can change significantly when the map is shifted or rotated). However, the PM quadtree is less prone to suffer from this problem than are other quadtrees. For example, shifting or rotating the region quadtree and the line quadtree [18] can lead to much larger changes in storage requirements. In contrast, the PM_2 and PM_3 quadtrees can be shifted or rotated without distortion or an unreasonable change in the storage requirements of the structure. Nevertheless, the storage requirements are still somewhat dependent on the positioning of the space within which the map is embedded. For example, the polygonal map of Figure 15a requires 7 PR quadtree leaf nodes. However, if we shift the map slightly, we get Figure 15b, which requires only 4 PR quadtree leaf nodes.

APPENDIX 3. Maintaining Labels of Regions

In the main body of the paper, we assume that the regions of a polygonal map are represented by labeling the borders of the regions with the region names. In

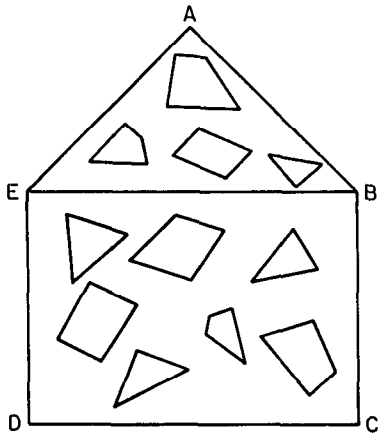


Fig. 16. Example polygonal map with many holes.

this appendix, we discuss how these labels can be efficiently maintained. To see the possible problems, let us consider Figure 16.

Figure 16 contains a pentagonal region ABCDE with many holes. All the q-edges forming the ABCDE border and those forming the outer border of each of the holes must be labeled with the name of the pentagonal region. Now consider what happens when edge BE is inserted. The outer border of the pentagonal region has been split in two and the new edge has been inserted forming two new regions—a triangular region and a quadrilateral region. Also the holes of the original pentagonal region must have their labels changed to indicate whether they are now in the triangular region or in the quadrilateral region. If the region labels were stored directly on each q-edge, then the updating of region labels would require that every q-edge be visited. If no additional data structure were used to organize the labels, then visiting each q-edge would be analogous to doing a connected component analysis of the portion of the map bounded by the pentagonal region. This analysis can be greatly simplified if all the q-edges in a particular region are kept in a linked list, but it will still involve an amount of work proportional to the number of q-edges in the outer border of the pentagonal region and in the outer border of all the holes inside the pentagonal region. In the following, we propose to organize the q-edges within a region in a more efficient manner than a linked list. Note that the above comments about the insertion of the edge BE apply equally to the problem of deleting the edge BE.

The q-edges that border a region can be partitioned into a collection of disconnected chains of q-edges. Elements of this collection correspond to walks along the outer border of the region, to walks along the outer borders of any holes in the region, and miscellaneous q-edges that have the same region on both their left and right sides (and hence are not properly borders of any region). The q-edges of each chain are grouped into a 2–3 tree [1] (termed a *chain* 2–3 tree) and we only store the label information (i.e., an identifier for the chain) in the root of the tree. In fact, each q-edge is linked to the leaves of at most two of

these 2–3 trees (one link for the region on each side of the q-edge) and the region information is found by moving up the tree using father links. Moreover, the chains of each region are also grouped into a 2–3 tree (termed a *region* 2–3 tree) and again we only store the label information in the root of the tree (this time the label information is the name of the region). The 2–3 tree is chosen because of its algorithmic simplicity (it is a degenerate case of the much studied B-tree [3]). In particular, the 2–3 tree has the property that insertion, deletion, and splitting and concatenation of two 2–3 trees can be performed in $O(\log_2 n)$ time for trees of size n. In addition, two 2–3 trees of size t_1 and t_2, respectively, can be merged into a new 2–3 tree in $O(t_1 + t_2)$ time. However, other balanced tree techniques could have also been used.

In order to be able to make use of the 2–3 trees we must be able to define an ordering for q-edges and chains. The q-edges in a given chain can be ordered according to their appearance during a walk along the chain starting at the leftmost of the uppermost vertices incident to the chain. In case of a tie (e.g., a closed chain in which case there are two q-edges meeting this criterion), we choose the q-edge whose other endpoint is uppermost leftmost. We also have to order the collection of chains corresponding to each region which we do by using their "extreme" q-edge. By extreme we mean that for each chain we choose the q-edge having the uppermost leftmost endpoint. Again, in case of a tie (e.g., extreme q-edges that share the same endpoint), we apply the same rule to the second endpoint. This extreme q-edge will provide a unique label for the entire region (with respect to a particular map).

If deletion or insertion of a q-edge does not change the number of regions in the map, then the operation can be performed in $O(\log_2 q)$ time where q is the number of q-edges (i.e., leaves) in the largest chain 2–3 tree associated with a given region. If deletion of a q-edge merges two regions, then the region 2–3 trees that order the collection of chains for each region will need to be merged. If the number of leaves (i.e., chains) in the region 2–3 trees representing these two regions is t_1 and t_2, respectively, then the cost of this operation will be $O(t_1 + t_2 + \log_2 q)$ time. This is accomplished by a procedure that merges the two collections of disconnected chains in $O(t_1 + t_2)$ time and creates the border of the new region from the borders of the two merged regions in $O(\log_2 q)$ time. Note that for cartographic data the individual chains are typically long (i.e., q can become large), but the number of chains in each region's collection is usually small for a given region (i.e., $t_1 + t_2$ is usually small). Indeed, if there are no chains that have the same region on both sides, then the average value of t_1 (and also t_2) could not be larger than 1 (corresponding to the region's outer border) plus the average number of holes. But since each hole is itself a region, the average number of holes cannot exceed 1. Therefore, the average value of t_1 (and of t_2) cannot exceed 2.

Similarly, when q-edge insertion causes splitting of a region with a region 2–3 tree of t chains, the cost will be $O(t + \log_2 q)$ time. This is accomplished by a procedure that separates the collection of disconnected chains of q-edges that bounded the old region (i.e., its region 2–3 tree) into a collection for each of the new regions (i.e., new region 2–3 trees) in $O(t)$ time and splits the outer border of the old region into the two outer borders of the new regions in $O(\log_2 q)$ time.

Note that the new region might be created by the inserted q-edge completing the border of a hole instead of connecting the outer border in 2 places. The cost of determining the region to which a q-edge belongs is $O(\log_2 q)$ time.

ACKNOWLEDGMENTS

We have benefited greatly from discussions with Randal C. Nelson and Clifford A. Shaffer. The suggestions of an anonymous referee are also appreciated.

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BENT, S. W., SLEATOR, D. D., AND TARJAN, R. E. Biased 2-3 trees. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science* (October, Syracuse, New York), IEEE, New York, 1980, pp. 248–254.
3. COMER, D. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–137.
4. EDELSBRUNNER, H. Key-problems and key-methods in computational geometry. In *Symposium of Theoretical Aspects of Computer Science* (April, Paris, France), Springer-Verlag, New York, 1984, pp. 1–13.
5. EDELSBRUNNER, H., GUIBAS, L. J., AND STOLFI, J. Optimal point location in a monotone subdivision. *SIAM J. Comput.* To appear.
6. FINKEL, R. A., AND BENTLEY, J. L. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (1974), 1–9.
7. HARARY, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.
8. HERTEL, S., AND MEHLHORN, K. Fast triangulation of simple polygons. In *Proceedings of the 1983 International FCT-Conference* (August, Borgholm, Sweden), Springer-Verlag, New York, 1983, pp. 207–218.
9. HUNTER, G. M. Efficient computation and data structures for graphics. Ph.D. dissertation, Deparment of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J., 1978.
10. HUNTER, G. M., AND STEIGLITZ, K. Operations on images using quad trees. *IEEE Trans. Pattern Anal. and Mach. Intell. 1*, 2 (Apr. 1979), 145–153.
11. KIRKPATRICK, D. Optimal search in planar subdivisions. *SIAM J. Comput. 12*, 1 (Feb. 1983), 28–35.
12. KLINGER, A. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, Ed. Academic Press, New York, 1971, pp. 303–337.
13. MARTIN, J. J. Organization of geographical data with quad trees and least square approximation. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing* (June, Las Vegas), IEEE, New York, 1982, pp. 458–463.
14. NIEVERGELT, J., AND PREPARATA, F. P. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM 25*, 10 (Oct. 1982), 739–747.
15. ORENSTEIN, J. A. Multidimensional tries used for associative searching. *Inf. Process. Lett. 14*, 4 (June 1982), 150–157.
16. OVERMARS, J. H., AND VAN LEEUWEN, J. Dynamic multi-dimensional data structures based on quad- and *k-d* trees. *Acta Informatica 17*, (1982), 267–287.
17. SAMET, H., AND WEBBER, R. E. Using quadtrees to represent polygonal maps. In *Proceedings of Computer Vision and Pattern Recognition 83* (June, Washington, D.C.), IEEE, New York, 1983, pp. 127–132 (also University of Maryland, Computer Science Dept. TR-1372).
18. SAMET, H., AND WEBBER, R. E. On encoding boundaries with quadtrees. *IEEE Trans. Pattern Anal. and Mach. Intell. 6*, 3 (May 1984), 365–369.
19. SAMET, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (June 1984), 187–260.

20. SAMET, H., ROSENFELD, A., SHAFFER, C. A., AND WEBBER, R. E. A geographic information system using quadtrees. *Pattern Recog. 17*, 6 (Nov./Dec. 1984), 647–656.
21. SHNEIER, M. Two hierarchical linear feature representations: Edge pyramids and edge quadtrees. *Comput. Graph. and Image Process. 17*, 3 (Nov. 1981), 211–224.
22. TAMMINEN, M. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34, Helsinki, 1981.
23. WEBBER, R. E. Analysis of quadtree algorithms. Ph.D. dissertation, TR-1376, Computer Science Department, University of Maryland, College Park, Md, March 1984.

Received November 1983; revised September 1985; accepted September 1985.