Using linear quadtrees to store vector data *

Hanan Samet
Clifford A. Shaffer

Computer Science Department and
Center for Automation Research
University of Maryland
College Park, Maryland 20742

Robert E. Webber

Computer Science Department
Rutgers University, Busch Campus
New Brunswick, New Jersey 08903

ABSTRACT

The linear quadtree is adapted to store vector data by defining a new data structure called a *segment quadtree*. It uses a constant or bounded, amount of storage per node, represents straight lines exactly (i.e., it is not a digitized representation), and enables updates in a consistent manner (i.e., when a vector feature is deleted, the database can be restored to the state it would have been in had the deleted feature never been inserted). The segment quadtree is shown to meet these requirements whereas existing quadtree-like methods (e.g., the edge quadtree, strip tree, etc.) fail to satisfy them. In order to illustrate the usefulness of the segment quadtree, sample algorithms are discussed to insert and delete line segments as well as perform boundary following. The space requirements of segment quadtrees are also investigated.

1. Introduction

The region quadtree representation [Klin71, Same84a] has gained extensive use as a data structure for region representation in image processing, computer graphics, automated cartography, and other fields. It is a hierarchical data structure that is based on the principle of regular decomposition. Due to the large amounts of data involved in such applications, it is useful to use a representation for the quadtree

* This work was supported in part by the National Science Foundation under Grant DCR-83-02118 and in part by the U.S. Army Engineer Topographic Laboratory under contract 70-81-C-0059.

that does not involve pointers. The linear quadtree [Garg82] is one such representation that stores the image as a collection of leaf nodes each of which is encoded by a number corresponding to a sequence of directional codes that locate the leaf along a path from the root of the quadtree.

We are interested in using the linear quadtree as a uniform representation for data corresponding to regions, points, and vector features. The uniformity facilitates the performance of set operations such as intersecting a vector feature with an area, etc. Use of a linear quadtree for point and region data is well understood; however, this is not the case for vector features. While there are several hierarchical data structures for vector features, they are either not based on regular decomposition (e.g., the strip tree [Ball81]), do not handle more than one vector feature (e.g., the strip tree), or do not cope with the intersection of vector features (e.g., the edge quadtree [Shne81]). For vector features, a good linear quadtree representation must also have the following three properties. First, it must use a constant, or bounded, amount of storage per node. This rules out the PM quadtree [Same85b]. Second, straight line segments should be represented exactly (not a digitized representation). Third, updates must be consistent, i.e., when a vector feature is deleted, the data base must be capable of being restored to a state identical (not an approximation) to that which would have resulted if the deleted vector feature had never been added. The line quadtree [Same84b] is eliminated because it only handles rectilinear vector features. Hunter and Steiglitz's [Hunt79] adaptation of the quadtree for polygons is likewise of no use since it only approximates the lines. The edge quadtree [Shne81] was designed primarily for representing curves and thus does not handle connected edges in a manner that permits consistent updates.

In order to meet the above requirements, we present a new data structure, termed a *segment quadtree*. We show how it can be used in conjunction with linear quadtrees to represent collections of vector features consisting of sets of connected straight line segments termed *polygonal maps*. The remainder of this paper is organized as follows. Section 2 reviews the region quadtree and other attempts to use quadtree-like data structures for storing vector feature data. Section 3 describes the segment quadtree while Sections 4, 5, and 6 show, respectively, how line segments are inserted into segment quadtrees, how they are deleted, and how a boundary represented by a segment quadtree is followed. Section 7 contains a worst-case analysis of the size and storage requirements of segment quadtrees as well as some empirical data. Section 8 summarizes the current state of, and future plans for, the investigation of the segment quadtree.

2. The quadtree and previous line representations

The region quadtree represents region data by successive subdivision of the image array into four isomorphic quadrants. Given an image and its binary array, if the array is not composed entirely of 1's or entirely of 0's, then we subdivide it into quadrants, subquadrants, ..., until we obtain square blocks (possibly single pixels) that consist entirely of 1's or entirely of 0's. As an example, the region in Figure 1a is represented by the 2^3 by 2^3 binary array in Figure 1b. The resulting blocks for the array of Figure 1b are shown in Figure 1c and the quadtree in Figure 1d.

Below we
issues we
node of d
the point
However,
this is no

The PR q
dle point
into quad
The colle
2b.

As mentio
regions, a
for vecto
requireme
reviewed

In the *ed*
part there
tains at m
Each leaf
it: magnit
grey-scale
resulting
such as le
flag is set

Applying
in a few l
intersectin
course, m
intersected
edge quad
polygonal

A serious
two or m
to an edge
vertex - e.
lowing as
vicinity of
which is c
Steiglitz [H
a region as
shows the
quadtree h
Again, a ve
tion of lin
quadtree o

Below we consider quadtree representations for line and point data. One of the first issues we must face is the location of a point specified with d bits of precision in a node of depth d. In order to simplify the theoretical analysis of Section 7, we treat the point as being located in the center of the node at level d in the quadtree. However, results analogous to that of Section 7 hold for other treatments. Thus this is not a restriction on the implementation.

The PR quadtree [Oren82, Same84a] is an adaptation of the region quadtree to handle point data. Given an image representing a set of points, the image is subdivided into quadrants, subquadrants, etc., until each quadrant contains at most one point. The collection of points in Figure 2a is represented by the PR quadtree of Figure 2b.

As mentioned in Section 1 our goal is a unified approach to the treatment of points, regions, and vector features. Thus we must find a similar decomposition criterion for vector features. Unfortunately, the strip tree [Ball81] does not meet our requirements as it is not based on a regular decomposition. Other candidates are reviewed below.

In the *edge quadtree* of Shneier [Shne81] a region containing a vector feature, or part thereof, is repeatedly subdivided into subquadrants until each quadrant contains at most one 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 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 the edge quadtree corresponding to the polygonal map of Figure 3 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. In other words, we don't know the nature of the vertex - e.g., the number of edges intersecting it. 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. Another quadtree variant which is closely related to the edge quadtree is the formulation of Hunter and Steiglitz [Hunt79], termed an *MX quadtree* in [Same84a]. 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 polygonal map of Figure 3. 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. Furthermore, note that the edge quadtree in Figure 4 contains considerably fewer nodes than the MX quadtree in Figure 5.

The *linear edge quadtree* is a variant of the edge quadtree that has been adapted for incorporation in a geographic information system [Same84c]. In this scheme, the leaf nodes of the quadtree are stored in a list (maintained in a B-tree) in the order in which they would have been visited by a preorder traversal of the tree. Each node contains three fields; an address, a type, and a value field. The address field describes the size of the node and the coordinates of one of the corners of its corresponding block. The type field indicates whether the node is empty (i.e., WHITE), contains a single point, or contains a line segment. Unlike Shneier's [Shne81] formulation, a line segment may not end within a node since in the existing implementation the value field is not large enough to contain the location of an interior point as well as a slope. Thus endpoints and intersection points are represented by single pixel-sized point nodes. The value field of a line segment indicates the coordinates of its intercepts with the borders of its containing node. Vertices are represented by pixel-sized nodes with the degree of the vertex stored in the value field. Figure 6 illustrates the linear edge quadtree representation. Note the difference in the decomposition of the region containing the vertex H in Figures 4 and 6.

The linear edge quadtree has a number of deficiencies. All vertices and endpoints are stored at the lowest level of digitization, i.e., in nodes deep in the tree. There is no mechanism for following a line segment, as each node describes only that portion of a line segment which is contained within the borders of the node. In particular, given a node that contains a single point, there is no indication as to which of the neighbouring nodes are connected to the point by a line segment.

An important criterion for evaluating whether or not astorage representation handles line segments properly is if the successive insertion and removal of the same line segment leaves the map unchanged. Since the edge quadtree nodes store only local information, it is extremely difficult to restore nodes, by merging, which had been split apart by the original insertion. For example, it is not easy to determine the endpoints of the edges emanating from a given vertex. Thus over time, the representation's compactness could deteriorate until it becomes equivalent to the MX quadtree (i.e., line segments are represented by pixel-sized nodes).

An alternative approach to storing vector feature data is the PM quadtree [Same85b]. The PM quadtree evolved from a desire to adapt the PR quadtree to store a polygonal map in a manner which preserves the relationship between edges and vertices. In essence, whenever a group of line segments meet at a common point, those segments can be organized by the linear ordering derived from their orientation. Three variations of the PM quadtree, termed PM_1, PM_2, and PM_3, have been developed.

The PM_1 quadtree is based on a decomposition rule that permits more than one line segment to be stored at a node only if they meet at a vertex that lies within the borders of that node. Figure 7 shows the PM_1 quadtree corresponding to the polygonal map of Figure 3. From the decomposition of the line segments CD and CE, we observe that the representation of line segments which meet at narrow angles may require a large number of nodes.

The PM_2 quadtree permits more than one line segment to be stored at a node even when the vertex they share is not within the borders of that node. Figure 8 shows the PM_2 quadtree that corresponds to Figure 3. Note that the PM_2 quadtree requires fewer nodes than the PM_1 quadtree for the same map. However, we observe that when a line segment passes near a vertex that is not incident on it (e.g., segment DF passing near point E in Figure 8), it is possible that many nodes may be required to separate them.

The PM_3 quadtree is based on the same decomposition rule as the PR quadtree. All line segments that pass through the node are broken into a fixed number of separate groups. There is one group for all the lines that radiate from the vertex in the node. The remaining line segments are ordered according to the pair of sides of the node's containing block that they intersect. The group of line segments radiating from a vertex is organized by angular orientation and the remaining groups of line segments are organized by their intercepts with the side of the region represented by the node. Figure 9 is the PM_3 quadtree that corresponds to Figure 3. Note that the block containing vertex E has two line segments intersecting the vertex (i.e., EA and EC), and one line segment (i.e., DF) for the line segment intersecting the South and West boundaries of the block.

Although useful for storing polygonal maps in core, it is not easy to incorporate the PM_2 or PM_3 quadtrees into the fixed-width fields of the linear quadtree disk-based representation. The problem is that the amount of information stored at a single quadtree node varies widely. For example, in the PM_3 quadtree, a node can contain both a vertex and a set of line segments that do not pass through the vertex. The PM_2 quadtree represents an improvement in the sense that a node either corresponds to a vertex or a set of line segments but not both. The PM_1 quadtree limits a node further to correspond either to a vertex or to a single line segment. This is more compatible with the node size limitation posed by the linear quadtree. Indeed, the segment quadtree, presented in the next section, can be viewed as a linear quadtree adaptation of the PM_1 quadtree.

3. The segment quadtree

The segment quadtree has three types of nodes: 1)empty or WHITE, 2)vertex, and 3)line (i.e., a node containing a line segment whose endpoints are boundaries of the block corresponding to the node). A linear quadtree representation stores each node type in the same (fixed) amount of space. This space should be minimized since WHITE nodes will also be of the same size as line and vertex nodes, even though for them we need only to store information distinguishing them from line and vertex nodes.

A line node in the segment quadtree is defined to contain precisely one line segment which enters from one side and exits through another side. With each line node we store the coordinates of the vertices of the line of which it is a component. This enables us to determine easily whether or not two neighbouring line nodes are part of a larger line segment. Without this information, we cannot properly merge nodes after deletion of a nearby line segment. Note that recording the coordinates, although requiring more space, is preferable to recording the slope and intercept values for the line because it avoids precision errors as well as keeps all information in integer format.

We next consider the problem of storing vertex nodes. A vertex node in the segment quadtree is defined to contain precisely one vertex and no line segments which do not intersect the vertex. This is identical to the definition of the PM_1 quadtree. With each vertex node we store the *x* and *y* coordinates of the vertex that it represents.

In order to be able to perform boundary following, we must be able to follow line segments which extend from a vertex. This requires us to investigate the neighbouring nodes of the vertex node. There are three cases. First, if a neighbour is empty, then there are no line segments extending from the vertex in that direction. Second, if a neighbour contains a single line segment, then we can check the slope of that line segment to determine whether or not it intersects the vertex. Finally, if the neighbour is a vertex, then we must be able to detect whether or not there exists an edge joining the two vertices. The solution is to store an eight bit descriptor with each vertex node that indicates which of the sides and corners are exited by one or more line segments. Using this scheme, whenever two vertices are contained in adjacent nodes, one of three situations can arise:

(1) The corresponding sides (or corners) are not exited through;

(2) the corresponding sides (or corners) are exited through; and

(3) the side for one node is marked as exited through, and the corresponding side of the other node is not exited through.

(1) indicates that no line segment joins the two vertices (e.g., the boundary between vertices B and C in Figure 10), while (2) indicates that a line segment does join the vertices (e.g., the boundary between vertices A and B in Figure 10). (3) signifies that no line segment passes between the two vertices and that the vertex node with the exited side is larger than the vertex node with the unexited side (since there must be another node on that side into which a line segment exited). For example, consider vertices B and D in Figure 10. Note that all the nodes neighbouring a given node on an exited side must be inspected since more than one line segment may exit from that side (e.g., the East side of the node containing vertex D in Figure 10).

In summary, segment quadtree nodes may contain either a vertex in which case each line segment in the node intersects the vertex, or at most a single line segment which enters and exits the node. The resulting decomposition is identical to that of the PM_1 quadtree; however, the information stored is different. Figure 11 shows the segment quadtree corresponding to the polygonal map of Figure 3. Compare this with the PM_1 quadtree of Figure 7.

An important aspect of the above decomposition rule is that it does not allow any case where there might be an unbounded decomposition of the tree. This would result if we would have taken as our decomposition rule one that did not permit a quadrant to contain more than one line segment. In the segment quadtree all vertices must be specified at some level of resolution, which will correspond to some depth in the tree. This resolution, say d, is a function of the minimum separation between any two vertices. The maximum depth of the segment quadtree depends on two further factors: the minimum separation between a vertex and a line and the minimum separation between two lines. These factors are analyzed further in Section 7.

In the case of geographic data, images are constructed from sequences of line segments that intersect only at their endpoints. Note that if a user wishes to specify two intersecting line segments, then four line segments must be specified. In the following, we assume that this minor preprocessing of the data has already been done.

4. Insertion into the segment quadtree

The data model used by the above analysis assumes that line segments do not intersect (except at their endpoints). This model is appropriate for cartographic data; however, the data structure is not restricted to such data. Below we show how to handle intersecting line segments by creating vertex points at the intersection of a collection of line segments. We shall also use this extended model in the discussion of deletion of line segments from the segment quadtree.

Insertion of a line segment into a segment quadtree proceeds as follows. If the line segment is to be inserted into a region represented by an internal node of the quadtree, then it is clipped against each of the quadrants of that region and the appropriate component of the line segment is inserted into the corresponding subquadrants. Once a line segment has been clipped, it is important to remember whether its end points were vertices of the original line segment or artifacts of the clipping operation.

Upon encountering a leaf node there are three possible courses of action depending on its type - i.e., empty, line, or vertex. If the leaf is empty and the component of the line segment contains two vertices (i.e., it is the unclipped line segment), then the region is further subdivided. If the leaf is empty and the component of the line segment contains one vertex, then the node becomes a vertex node and is marked exited in the direction of the intercept of the line segment with the border of the node. If the component of the line segment contains no vertices, then the node becomes a line node.

If the leaf is a line node, and the line segment to be inserted contains any vertices, then the node must be decomposed. Otherwise, we must determine if the components of the line segment and the line represented by the line node intersect within the region bounded by the node. If they intersect, then a new vertex is formed at the intersection point with the borders of the node being marked as exited as appropriate for the four line segments radiating from that vertex. Otherwise, the nodes are decomposed further until the two line segments are not contained within the same node.

If the leaf is a vertex node, and the component of the line segment either contains another vertex or does not intersect the vertex, then the leaf node must be further decomposed and the line segment insertion procedure continues. If the component of the line segment intersects the vertex and contains no new vertex, then we need only mark the edges of the node that intersect the line segment as exited.

5. Deletion from the segment quadtree

Deletion of a line segment from a segment quadtree is achieved by applying a two-step process to each quadtree node through which the line segment passes. First, we must remove the information corresponding to the presence of the line segment. Second, we must merge empty nodes as well as nodes that correspond to a line segment once the nodes containing the removed vertex have been replaced by empty nodes.

The first step of removing the information corresponding to the presence of a line segment is quite easy. We simply repeat the decomposition of the line segment as though it were to be inserted, locating those segment quadtree leaf nodes which contain the segment. Line nodes are marked WHITE. Vertex nodes require more work since the sides of the containing leaf through which the segment exits may also be exited by another (undeleted) line segment. Therefore, the neighbours along these sides must be examined to see if they contain another line segment. If not, then the vertex segment bit marking that side as exited is turned off. If, after this process, no side of the node's block remains marked as exited, then the line segment being deleted is the only one intersecting that vertex, and the node is marked WHITE.

Merging eligible nodes is somewhat more complicated; but since it is possible to determine which neighbours of a node are connected by a line segment, it is feasible. Each node which contained a portion of the line segment being deleted must be considered as a potential candidate for merger with its three siblings. The following algorithm should be applied to each such node.

A node and its three siblings may be merged if they are either 1) all WHITE, 2) all line nodes representing portions of the same line, or 3) a single vertex with line segments all of which intersect the vertex. The first two cases are easily determined by comparing the values of the node and its siblings - the siblings in this case will all be at the same depth in the tree. The third case might be difficult to determine, since two line segments may intersect at a vertex which either is not contained within the region covered by the node and its siblings, or is stored at a lower level in one of its sibling's subtrees.

In order to handle this last case, it is necessary to check the subtrees of each sibling of the node to determine if all line segments stored intersect at the same vertex. If the vertex is not contained in these siblings, then we must check the siblings of the node's ancestors until either 1) the vertex is located, or 2) additional line segments or vertices are encountered which would not allow the nodes to merge. Note that a special situation arises when exactly two line segments meet at

a vertex
merged t
would ha

For exam
which is
since it n
tex E is
deleting s
leaf node

6. Trave

In order
accomplis
(having s
in questic
using the

If a line
line node
found (in
that verte
the minim
traversal

At each st
node. If
new neigh
with the i
directions
node excep

If we are
in a count
segment r
found is t
the polygo
ple, suppo
we are mo
we must
rather thar

7. Space r

Earlier wo
terms of t
segments i
the case t

a vertex and they are collinear. In such a situation, the appropriate nodes are merged to form a line node instead of being merged to form a vertex node as would have otherwise happened in the third case.

For example, consider the deletion of line segment AE from Figure 3, the result of which is shown in Figure 12a. The leaf node marked I is the result of merging since it now contains only portions of line segment AG. The node containing vertex E is no longer marked as exited on the North. Figure 12b shows the result of deleting segment CE from Figure 12a. In the course of deleting the line segment all leaf nodes in the SE quadrant will be merged to form the node marked II.

6. Traversing the border of a polygon

In order to traverse the border of a polygon we must first locate it. This can be accomplished, for example, by visiting successive western neighbours of a node (having started with some leaf representing a region that is internal to the polygon in question) until either a line or a vertex node is reached. This can be achieved using the neighbour finding techniques described in [Same82, Same85a].

If a line node is reached, then proceed down the line segment represented by the line node such that the interior of the polygon is on the right. If a vertex node was found (instead of a line node), then examine all the line segments radiating from that vertex in the counterclockwise direction looking for the segment that forms the minimum interior angle with it. We are now positioned to perform a clockwise traversal of the perimeter of the polygon.

At each step of the traversal, we are either entering a line segment node or a vertex node. If we are entering a line segment node, then all we must do is calculate the new neighbour. The information in the line node should be checked for consistency with the information used to calculate the entry into the line node. However, since directions only change at vertex nodes, nothing new is learned from a line segment node except that the border hasn't changed direction yet.

If we are entering a vertex node, then the neighbours of the vertex node are visited in a counter-clockwise manner (from the node we enter from) until the next line segment radiating from the vertex is located. This insures that the line segment found is the next line segment met in the clockwise traversal of the perimeter of the polygon (in case the vertex is intersected by many line segments). For example, suppose we are following the border of the polygon ABCEA in Figure 11 and we are moving along line segment BC in the direction of C. Upon reaching vertex C, we must search in a counter-clockwise direction in order to locate segment CE, rather than segment CD which is part of an adjacent polygon.

7. Space requirements

Earlier work [Same85b] on PM quadtrees analyzed the storage requirements in terms of the distances between vertices in the map and the angles formed by line segments in the map. Although such an analysis is technically correct, it is seldom the case that we categorize a map in such terms. A more common way of describing a map is in terms of the precision with which the locations of the vertices of the map are measured. We shall use this technique in the following analysis of the PM_1 quadtree (which is equivalent to the analysis of the segment quadtree).

As has been observed elsewhere [Same85b], the key factor in the analysis of the storage requirements of the PM_1 quadtree is the maximum depth of the quadtree. Given the maximum depth, say d, it is clear that the number of quadtree nodes representing a polygon will be proportional to 2^d (except in certain degenerate cases). This follows from the analysis of the MX quadtree performed by Hunter [Hunt79] and is definitely superior to using a 2-dimensional array to store the same information with a corresponding storage requirement of 4^d. Of course, PM quadtrees are usually considerably more compact than MX quadtrees (as will be demonstrated in the empirical results at the end of this section). However, this is not revealed by our approach to the worst-case analysis of these structures.

Analyzing the worst-case storage requirements in terms of the maximum depth is particularly useful when contemplating a linear quadtree implementation because efficiency considerations require that the linear quadtree address be stored in an array whose elements have a fixed and bounded size. Each element of this array will have a width, in bits, equal to twice the maximum depth of the quadtree being stored. Hence the following analysis will yield a safe size for the linear quadtree address as a function of the number of bits used to store the coordinates of the map.

The situation that leads to a maximum depth for vertices stored with d bits of precision can be derived in the following manner. Observe that all the points that can be specified with d bits of precision form a 2^d by 2^d grid. Recall that the PM_1 (and segment) quadtree require that no node can contain two line segments unless they meet at a common vertex in the region of that node (here we can view an isolated vertex as a degenerate line segment). Thus the problem reduces to how small (or deep) a node can be and still fail to satisfy this requirement. First, we determine the closest approach (without touching) between a point, say C, on that grid and a line segment, say AF, connecting two other points (A and F) on that grid. Assuming that our grid of points corresponds to centers of pixels, we can see that this has placed the vertex C at the maximum depth for a map of one line segment and an isolated vertex. Now we find two other points, say W and Y, such that the angle formed by WCY is a minimum. Let W' be the intercept of the border of the node containing C with the line segment CW and let Y' be the intercept of the border of the node containing C with the line segment CY. We claim that on a map where all line segment intersections are represented by vertices with precision d, that the maximum depth of the corresponding PM_1 (or segment) quadtree is the depth necessary to separate W' from Y'. Figures 13 and 14 show the worst case positioning of A, C, F, W, and Y. In the following discussion we demonstrate that this is indeed the worst-case situation and calculate the corresponding worst-case depth.

First, let us consider how close C can approach AF. If we drop straight down from C onto the segment AF, we find a point G on AF. Although the distance between C and AF is not exactly the length of CG, AF is no closer to C than $1/\sqrt{2}$ times the length of CG. To find the length of CG, we argue from the similar triangles ACG

and AF
ratio of
HF) is
approxi

To see
isolated
observe
both) is
culated
node of
that to
tree add

First, ob
than d
an endp
from ve
with H
be irrele
represen
what le
non-inte
deeper is
nodes on
line segm
and W',
lest possi

In order
introduce
length).
on a stra
lengths o
Observe
Y''Q is t
length of
triangles
is the sam
WCQ and
the ratio
ratio of Q
QC is 2^{d-}
that node
tions, we

We can n
$2^{-(3d+1)}$
to separat
approach
depth of

and AHF that the ratio of the length of CG to the length of HF is the same as the ratio of the length of AC to the length of AH. If we assume that AC (and hence HF) is of unit length, then the length of AH is $2^d - 1$ and the length of CG is approximately 2^{-d}.

To see that this is the worst-case situation for a map of one line segment and an isolated vertex, we consider all the other possible combinations of points and observe that when we form the analogous similar triangles, either AC or HF (or both) is greater than 1 and hence the length of CG would be greater than that calculated above. If this was as far as the analysis went, then we would see that a node of width 2^{-d}, occurring at depth $2d$, would be sufficient. This would mean that to store vertices with d bits of precision in each coordinate, the linear quadtree address field must have room for $4d$ bits.

First, observe that the point G is not a vertex of our map (and would require more than d bits of precision to represent). Therefore, line segments that contain G as an endpoint are irrelevant to our analysis since we are looking at maps formed from vertices of the grid. Similarly, if a point below A (say B) were connected with H resulting in an intersection point, say D, with the line AF, this would also be irrelevant. This is because D would be a vertex of the map, but would not be representable in d bits of precision. However, by the above analysis we now know what level of decomposition is necessary to separate vertex C from its nearest non-intersecting line segment. Therefore, the only way we can cause nodes to be deeper is to realize that any line segments connected to C have to be in separate nodes once they leave the node containing C. The closest approach between two line segments connected to C, say CY and CW, occurs at their intersection, say Y′ and W′, with the border of the node containing C. Thus, we must find the smallest possible angle YCW and then calculate the length of Y′W′.

In order to calculate the length of Y′W′ as shown in Figure 14, it is convenient to introduce a point Q such that WQC is a right angle (and hence WQ is of unit length). We shall also extend the line segment CY to CY″ where Y″, W, and Q lie on a straight line. Since triangles Y″WY and Y″QC are similar, the ratio of the lengths of Y″W and Y″Q is the same as the ratio of the lengths of WY and QC. Observe that the length of WY is 1 and the length of QC is $2^d - 2$. The length of Y″Q is the length of Y″W plus the length of WQ (which is 1). Therefore the length of Y″W is $1/(2^d - 3)$ (which is roughly 2^{-d}). Next, we observe that the triangles Y″CW and Y′CW′ are similar which means that the ratio of Y″W to Y′W′ is the same as the ratio of WC to W′C. Similarly, by observing that the triangles WCQ and W′CQ′ are similar, we find that the ratio of WC to W′C′ is the same as the ratio of QC to Q′C. Therefore, the ratio of Y″W to Y′W′ is the same as the ratio of QC to Q′C. Recall that the length of Y″W is $1/(2^d - 3)$, and the length of QC is $2^d - 2$. Since the depth of the node containing vertex C is $2d$, the width of that node must be 2^{-d} and the length of Q′C must be 2^{-d-1}. By solving the equations, we find that the length of Y′W′ is roughly $2^{-(3d+1)}$.

We can now complete our analysis by noting that the depth of a node of width $2^{-(3d+1)}$ would be $4d + 1$. However, the above analysis assumes that we are trying to separate Y′ from W′, whereas in reality we want to separate Y′Y from its closest approach W′W. This can be handled by using $4d + 2$ as an upper bound on the depth of the quadtree. Thus the linear quadtree address field should allow for

$8d+4$ bits. Since d is usually the wordsize, in bits, on the computer, using 9 words to store the linear quadtree address might seem a bit extravagant. However, it should be noted that our worst-case example was rather eccentric and that typical data behaves much better. Below, we cite some typical maximum depths and node counts for four maps from a cartographic database with which we have been working [Same84c]. The various strategies for handling maps that require greater depth than a user is willing to allocate are beyond the scope of this paper.

The above results are of interest when establishing the correctness of an implementation, i.e., they indicate the worst-case depth for which we must allow. However, this is one situation when the worst case is truly rare. As an example, let us consider four maps (Figures 15 through 18) chosen from our geographic database. The Railline Map (Figure 15) shows the path of a railroad through a section of the Russian River area in California. The Powerline Map (Figure 16) presents analogous information about the local main powerline. The Cityline Map (Figure 17) indicates the border of the local municipality. The Roadline Map (Figure 18) is our most complicated map, since it details the local roadway network. Table 1 contains the number of vertices and edges in each of these maps. All of these maps consist of line segments whose vertices rest on a 512 by 512 grid. Thus each vertex would require 18 bits of information (2 9-bit coordinates) to represent. The above analysis would indicate that we would have to be prepared for the possibility of the quadtree having a depth of 40. However, such a depth is not even approached by our sample data, and in general is extremely unlikely.

Table 1. Size of the maps.

Map	No. of Vertices	No. of Edges
Powerline	15	14
Railline	17	16
Cityline	65	64
Roadline	685	764

Tables 2-4 summarize the storage requirements of the MX, linear edge (recall Figure 6), and segment quadtrees (equivalent to the PM_1 quadtree) for the four maps of Figures 15-18. In all cases the MX quadtree is larger than the segment quadtree by at least a factor of 7. More generally, we would expect the size of the MX quadtree to be roughly as large as the product of the average line length and the number of nodes in the corresponding segment quadtree. In the case of trivial (but often typical) maps, like the Powerline, Railline, and Cityline, we see that the linear edge quadtree is almost three times as large as the segment quadtree. This is because the average depth of a vertex node in the segment quadtrees for these two maps was observed to lie between 6 and 7 whereas the linear edge quadtree had to represent all of the vertex nodes at depth 9.

Table 2: Size of the MX quadtrees.

Map	Depth	Leaves	BLACK nodes	WHITE nodes
Powerline	9	1627	526	1101
Railline	9	2074	680	1394
Cityline	9	2770	1187	1583
Roadline	9	20566	8955	11611

Table 3: Size of the linear edge quadtrees.

Map	Depth	Leaves	Vertex nodes	Line nodes	White nodes
Powerline	9	178	28	27	123
Railline	9	229	31	25	173
Cityline	9	542	133	71	388
Roadline	9	7723	1590	1354	4779

Table 4: Size of the segment quadtrees.

Map	Depth	Leaves	Vertex nodes	Line nodes	White nodes
Powerline	7	64	20	11	33
Railline	8	70	22	12	36
Cityline	8	220	90	31	99
Roadline	12	2701	1067	509	1125

In a map of moderate complexity, such as the Roadline Map (the most complex we have gathered to date) we see that the linear edge quadtree is a bit smaller than three times the size of the segment quadtree. For the first time, we observe the occurrence of a segment quadtree deeper than the depth required by the digitization grid. In this case the digitization grid required a depth of 9 and the segment quadtree actually had a depth of 12. Observe that this depth of 12 is still a long way from the worst-case value of 40 calculated in the previous analysis. Although for this map, the maximum depth of the segment quadtree is greater (i.e., 12) than that of the linear edge quadtree (i.e., 9), we must look to the distribution of nodes by depth (see Table 5) to explain the difference in the number of nodes between the two trees. In essence, the average depth of the vertex nodes is smaller for the segment quadtree than for the linear edge quadtree thereby accounting for the smaller number of nodes in the segment quadtree. The importance of the reduction in the average depth of the vertex nodes in the segment quadtree is a consequence of the the observation that the decomposition of a line segment is identical in the linear edge and segment quadtrees once the line segment has exited the region of the vertex nodes representing its endpoints.

Table 5
Distribution of node types by depth for the segment quadtree of the Roadline map.

Depth	Vertex nodes	Line nodes	White nodes
0	0	0	0
1	0	0	0
2	0	0	4
3	2	0	8
4	10	3	39
5	73	33	110
6	184	106	226
7	286	152	272
8	242	123	208
9	165	46	159
10	104	25	98
11	1	17	1
12	0	4	0

8. Conclusions

The segment quadtree has been presented and shown to be a suitable data structure for representing vector feature data in conjunction with image databases using linear quadtrees. Methods for insertion and deletion of line segments in it, as well as its use in border following have been described. In the segment quadtree, deletion of line segments and border following can be accomplished using information local to the nodes representing a line segment. In contrast, such operations in the linear edge quadtree are much more difficult. Furthermore, the segment quadtree requires fewer nodes than the edge quadtree to store a collection of vector features since the vertices need not be stored at the lowest level of resolution as in the edge quadtree. This means that fewer nodes will be required to represent corners and intersections. Future work includes its implementation and integration into the geographical information system described in [Same84c], where it will replace the linear edge quadtree.

References

[Aho74] - A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Ball81] - D.H. Ballard, Strip trees: A hierarchical representation for curves, *Communications of the ACM 24*, 5(May 1981), 310-321 (see also corrigendum, *Communications of the ACM 25*, 3(March 1982), 213).

[Garg82] - I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

[Hunt79] - G.M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

[Klin71] - A. Klinger, Patterns and search statistics, in *Optimizing Methods in Statistics*, J.S. Rustagi, ED., Academic Press, New York, 1971, 303-337.

[Oren82] - J.A. Orenstein, Multidimensional tries used for associative searching, *Information Processing Letters 14*, 4(June 1982), 150-157.

[Same82] - H. Samet, Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing 18*, 1(January 1982), 37-57.

[Same84a] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

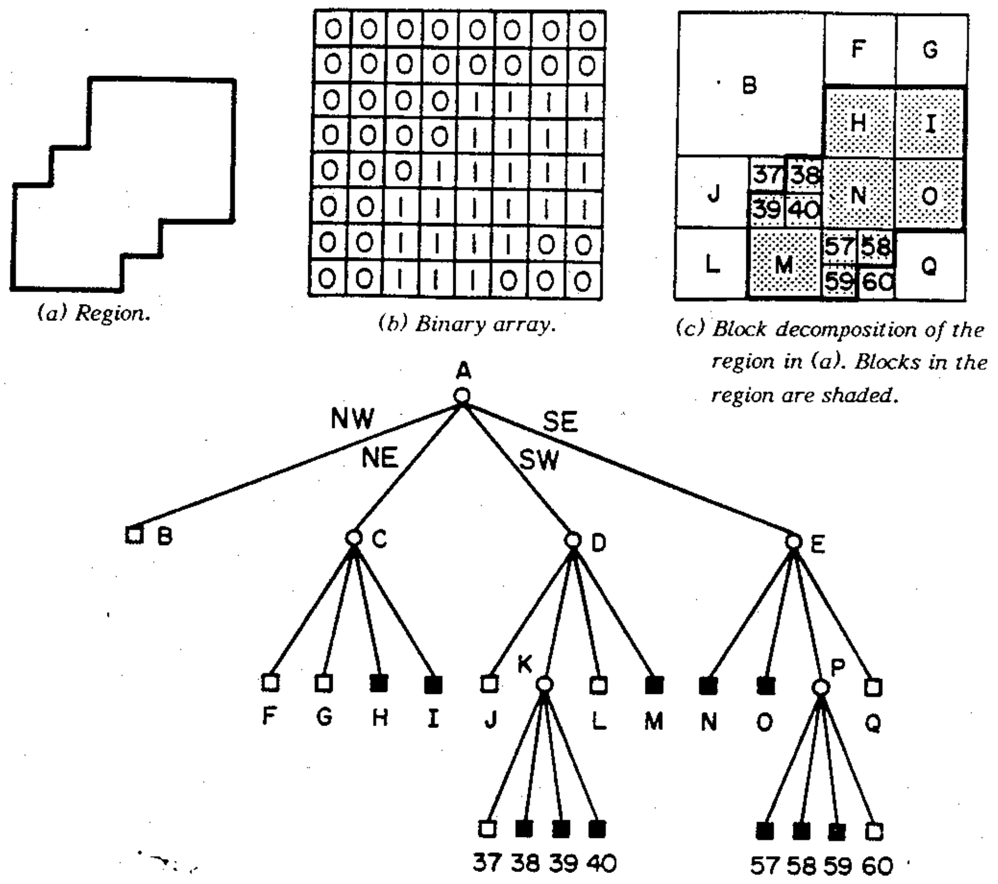
[Same84b] - H. Samet and R.E. Webber, On encoding boundaries with quadtrees, *IEEE Transactions on Pattern Analyg)c and Machine Intelligence 6*, 3(May 1984), 365-369.

[Same84c] - H. Samet, A. Rosenfeld, C.A. Shaffer, and R.E. Webber, A geographic information system using quadtrees, *Pattern Recognition 17*, 6 (November/December 1984), 647-656.

[Same85a] - H. Samet and C.A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, to appear in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1985 (see also University of Maryland Computer Science TR-1432).

[Same85b] - H. Samet and R. E. Webber, Storing a collection of polygons using quadtrees, to appear in *ACM Transactions on Graphics*, 1985 (see also *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, 127-132 and University of Maryland Computer Science TR-1372).

[Shne81] - M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Computer Graphics and Image Processing 17*, 3(November 1981), 211-224.



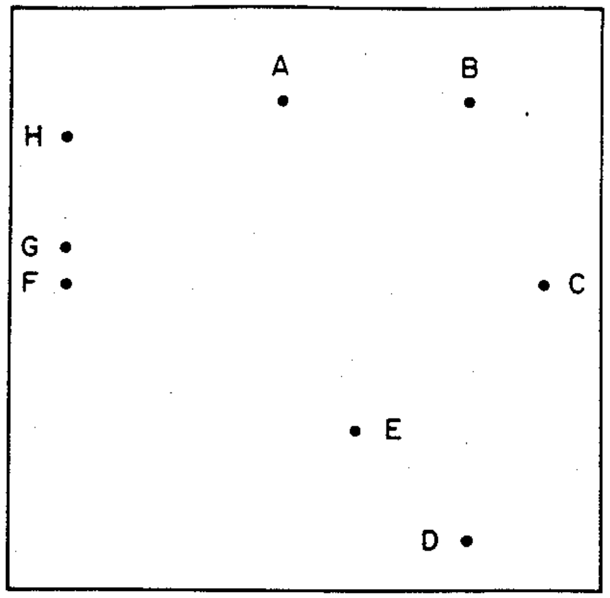
(a) Region.

(b) Binary array.

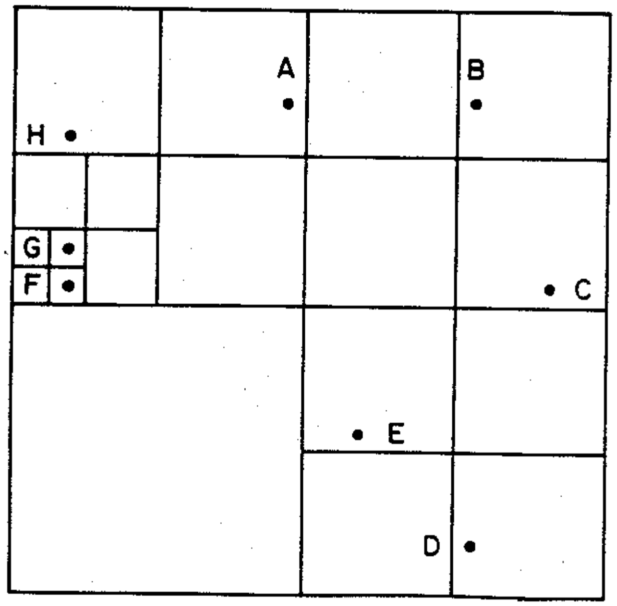
(c) Block decomposition of the region in (a). Blocks in the region are shaded.

(d) Quadtree representation of the blocks in (c).

Figure 1. A region, its binary array, and its maximal blocks.



(a) A collection of points.



(b) The PR quadtree for the points of (a).

Figure 2. A collection of points and its PR quadtree.

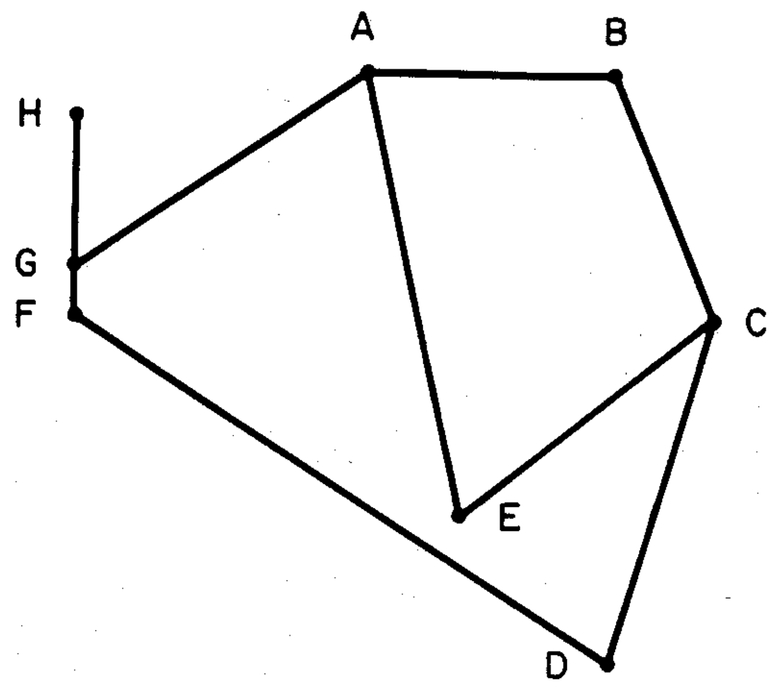


Figure 3. A polygonal map.

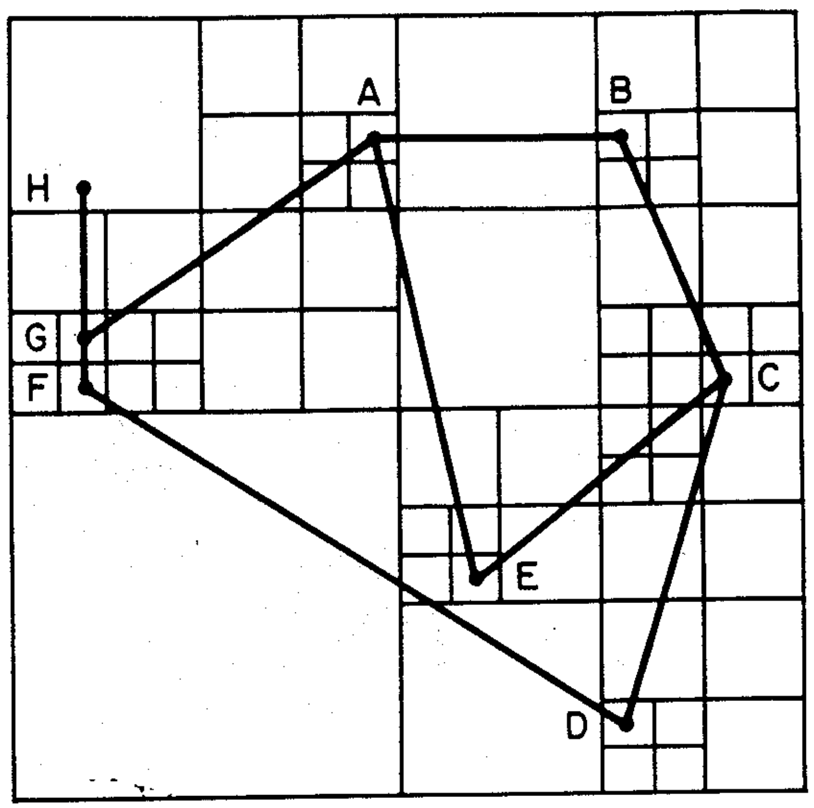


Figure 4. The edge quadtree for the polygonal map of Figure 3.

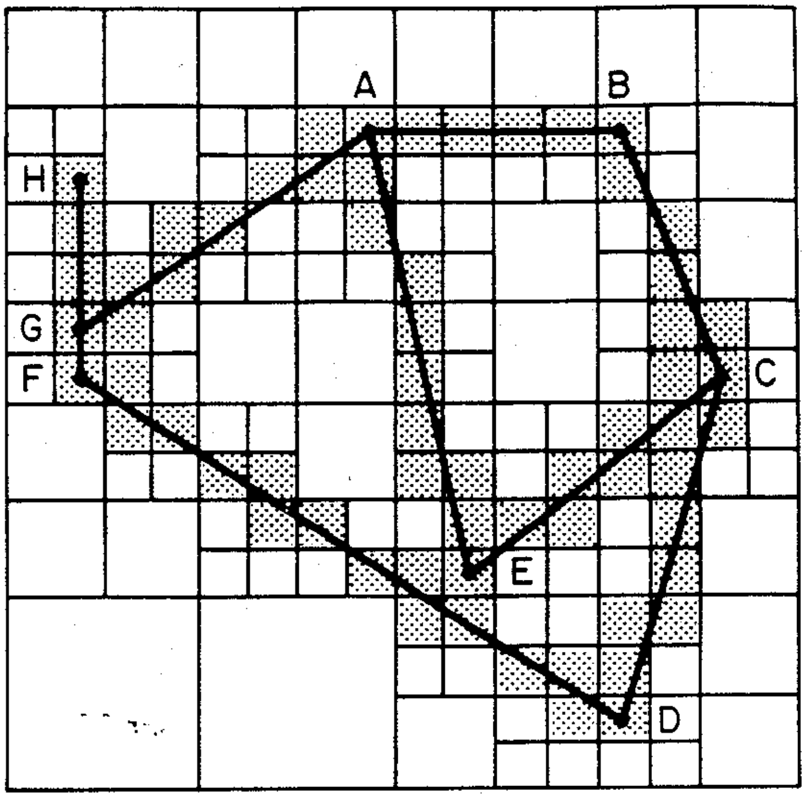


Figure 5. The MX quadtree for the polygonal map of Figure 3.

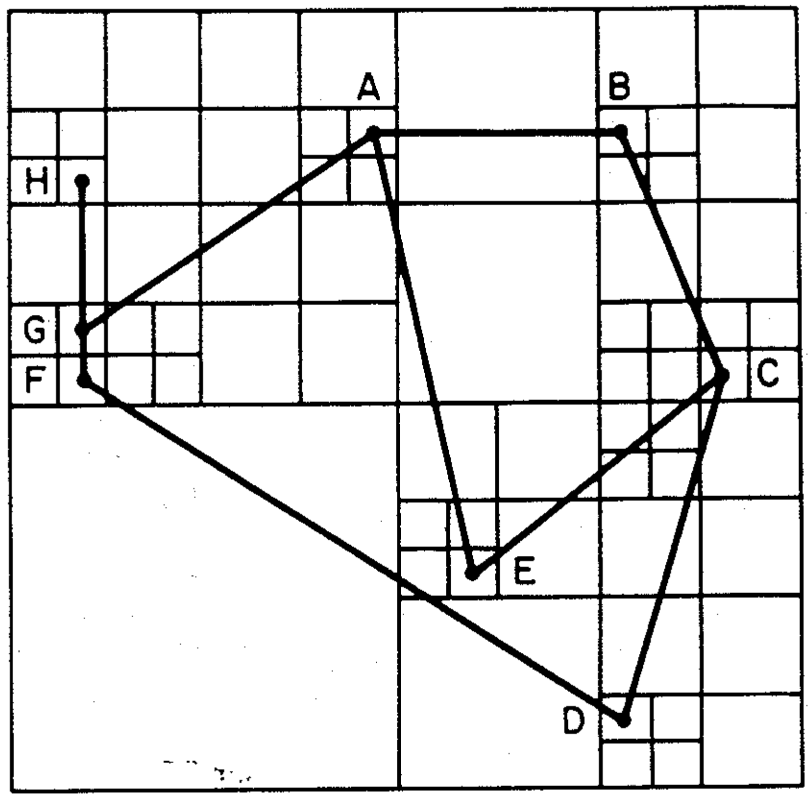


Figure 6. The linear edge quadtree for the polygonal map of Figure 3.

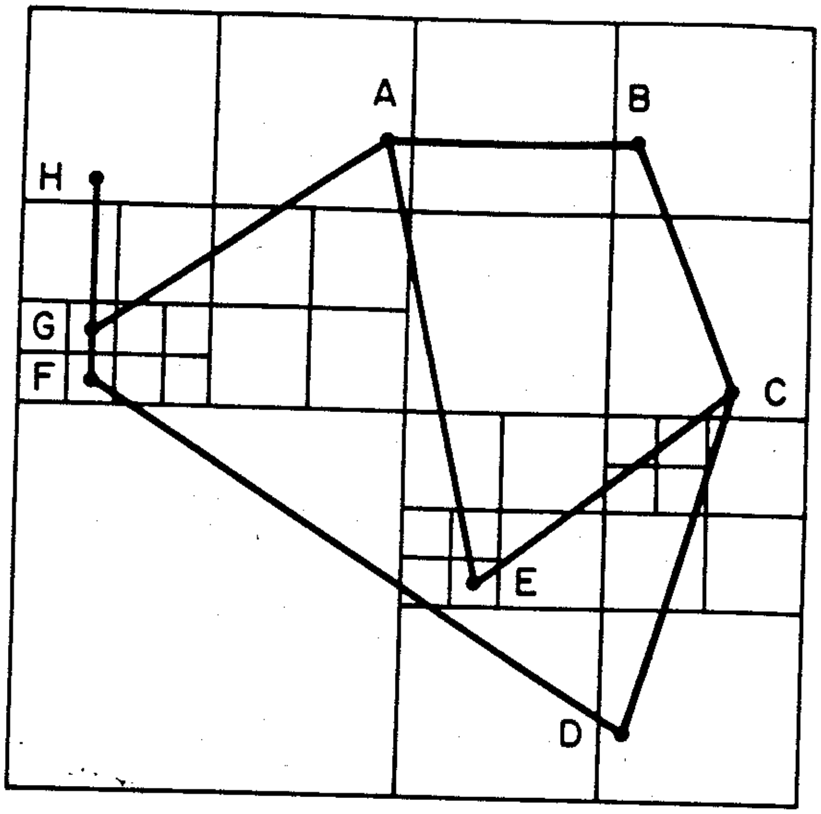


Figure 7. The PM$_1$ *quadtree for the polygonal map of Figure 3.*

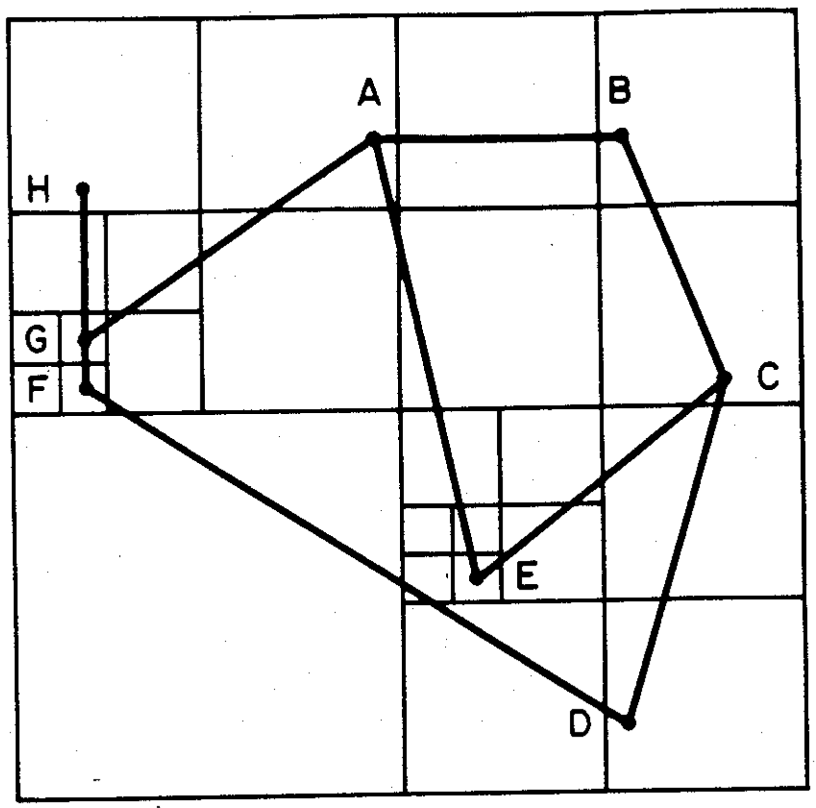


Figure 8. The PM$_2$ *quadtree for the polygonal map of Figure 3.*

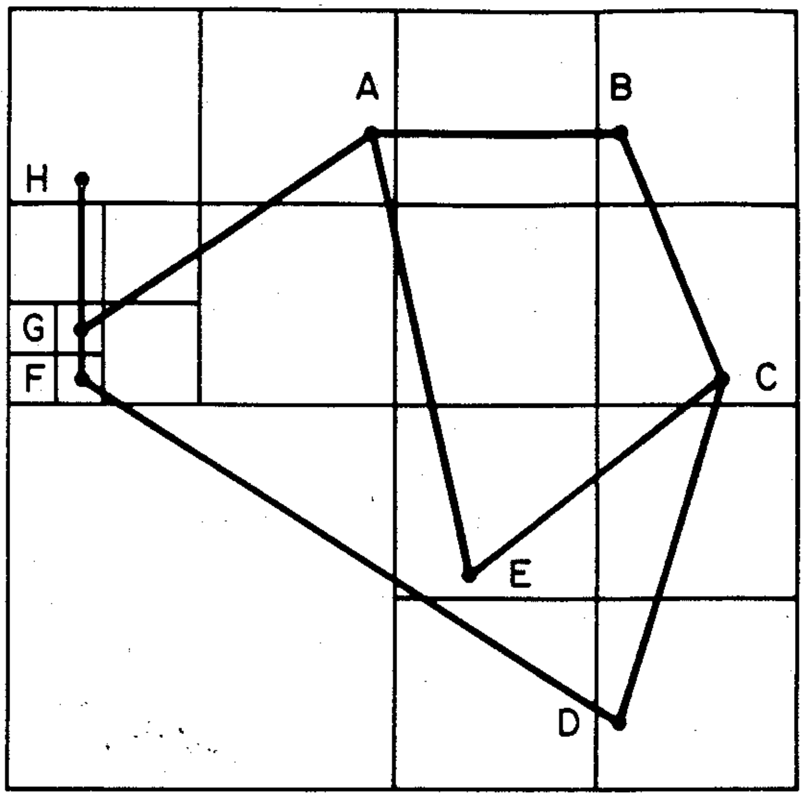


Figure 9. The PM$_3$ *quadtree for the polygonal map of Figure 3.*

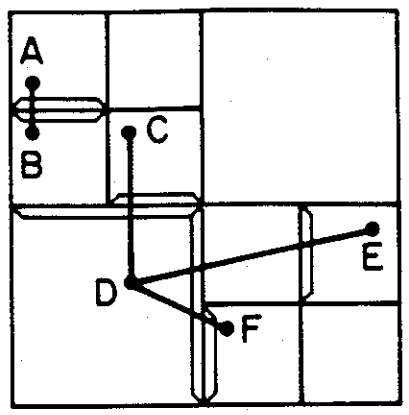


Figure 10.
The segment quadtree for a collection of line segments. Vertex nodes have exited edges marked by a bracket.

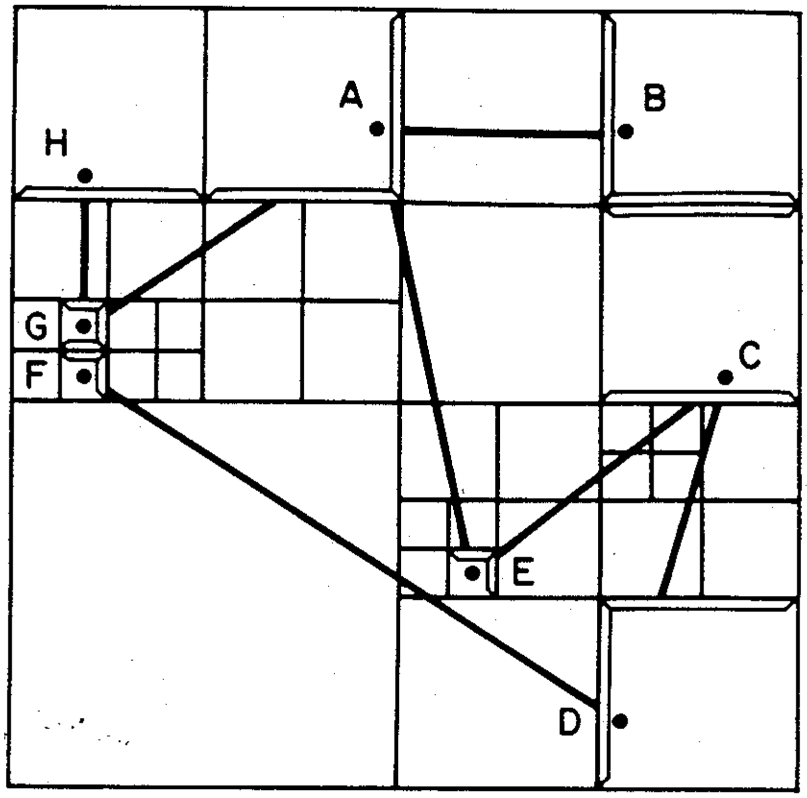
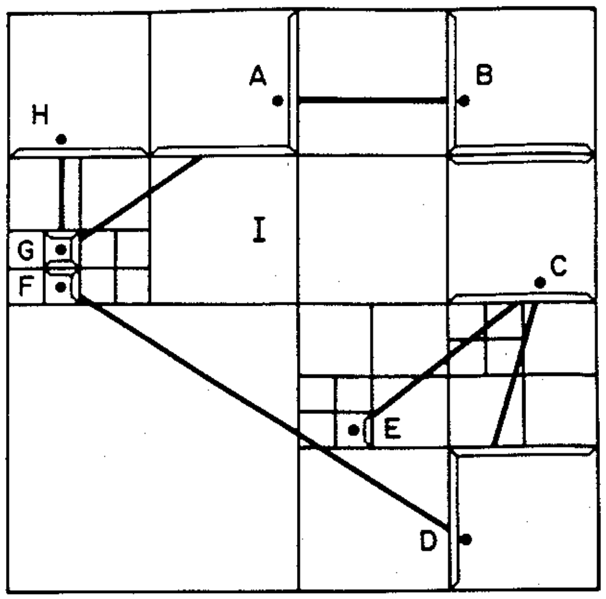
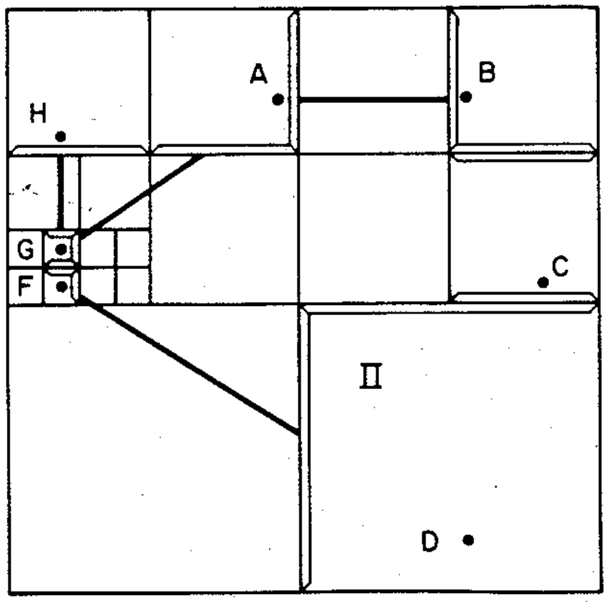


Figure 11. The segment quadtree for the polygonal map of Figure 3.



(a) The result of deleting segment AE from the segment quadtree of Figure 11.



(b) The result of deleting segment CE from the segment quadtree of (a).

Figure 12. Examples of deletion of line segments from the segment quadtree.

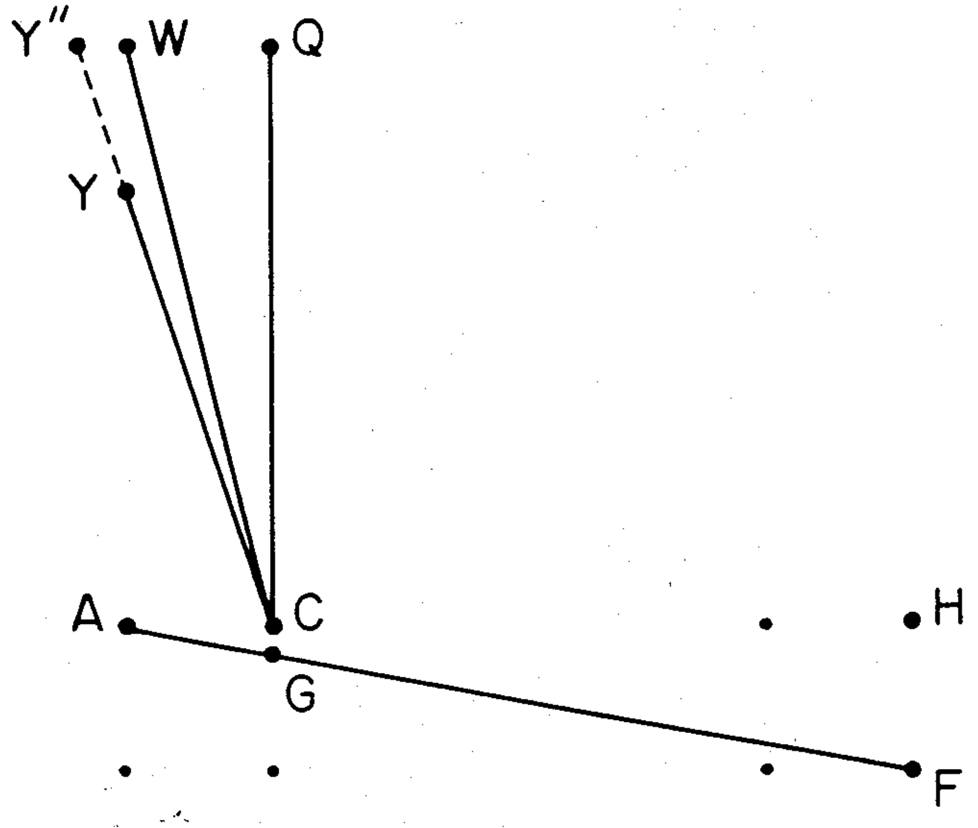


Figure 13. Map with worst-case storage requirements.

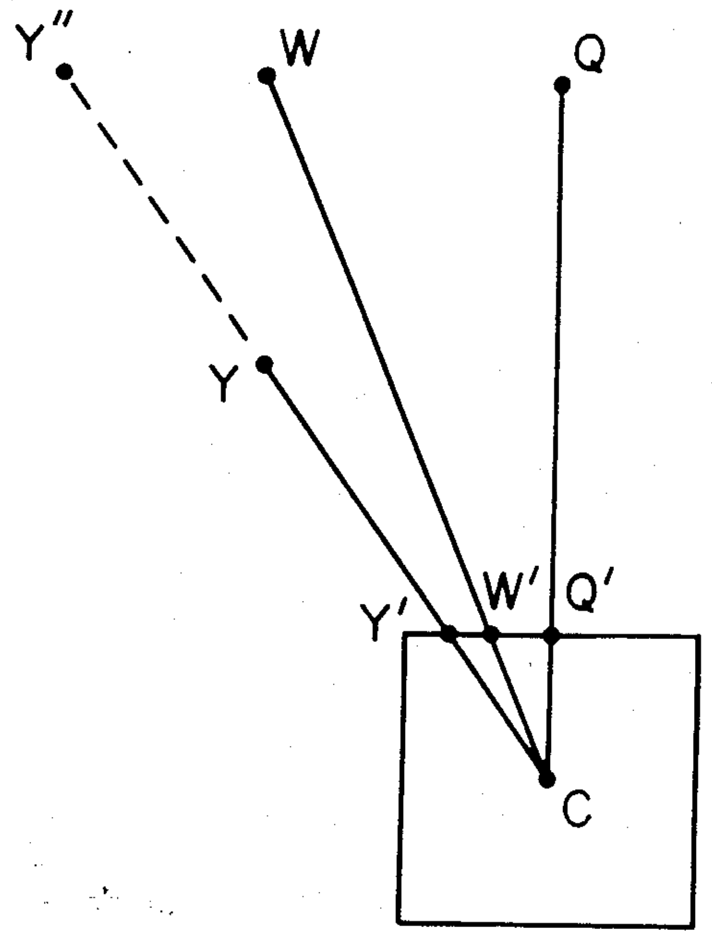


Figure 14. Detail of Figure 13 in the vicinity of point C.

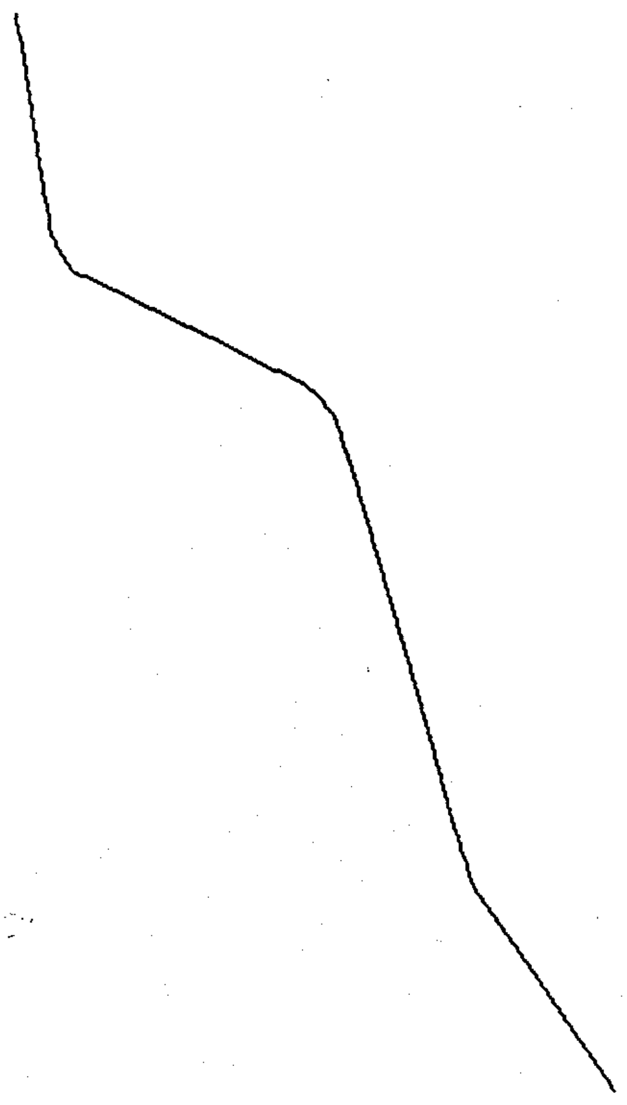


Figure 15. The Railline Map.

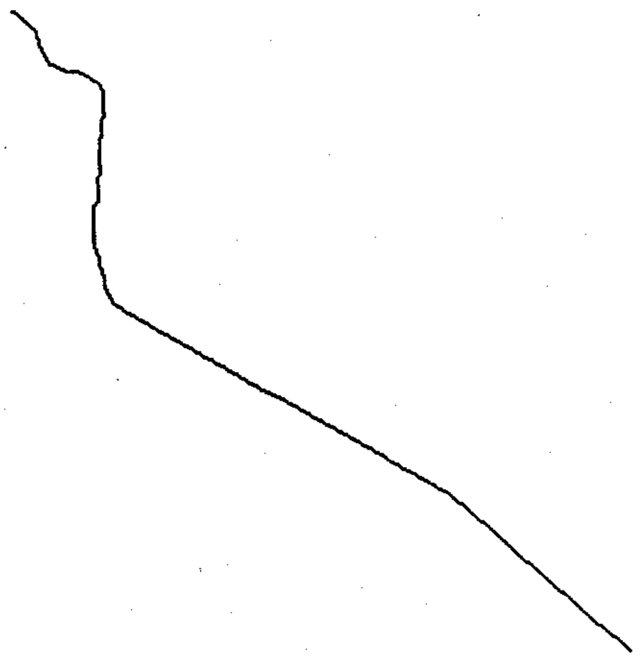


Figure 16. The Powerline Map.

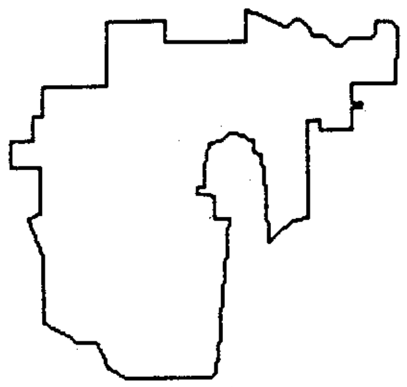


Figure 17. The Cityline Map.

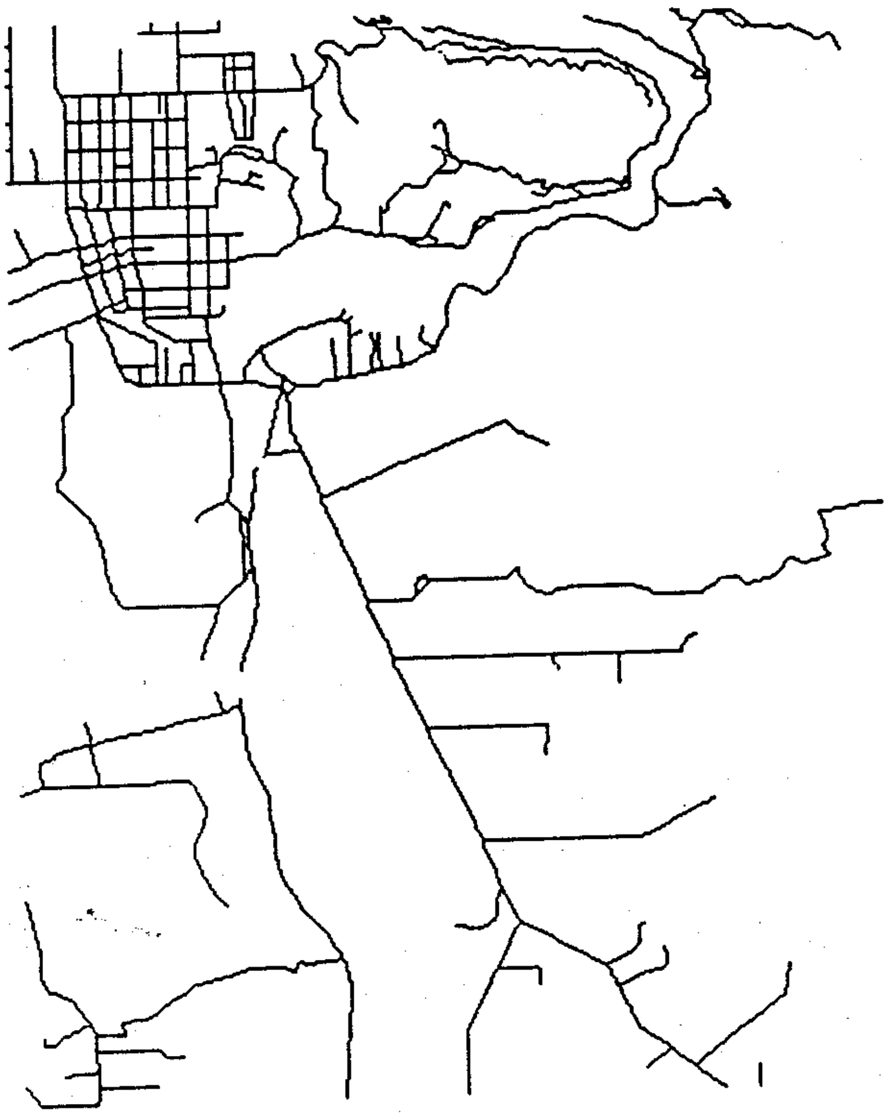


Figure 18. The Roadline Map.