A Consistent Hierarchical Representation for Vector Data

Randal C. Nelson
Hanan Samet

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

Abstract:

A consistent hierarchical data structure for the representation of vector data is presented. It makes use of a concept termed a *line segment fragment* to prevent data degradation under splitting or clipping of vector primitives. This means that the insertion and subsequent deletion (and vice versa) of a vector leaves the data unchanged. Vectors are represented exactly and not as digital approximations. The data is dynamically organized by use of simple probabilistic splitting and merging rules. The use of the structure for implementing a geographic information system is described. Algorithms for constructing and manipulating the structure are provided. Results of empirical tests comparing the structure to other representations in the literature are given.

CR Categories and Subject Descriptors: E.1 [**Data**]: Data Structures - trees; I.3.3 [**Computer Graphics**]: Picture/Image Generation - display algorithms; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling - object representations; geometric algorithms

General Terms: Algorithms, Data Structures

Additional Key Words and Phrases: vector data, quadtrees, hierarchical data structures, polygonal representations

I. Introduction

Quadtrees are a useful structure for representing certain types of geometric or geographic data. In particular, point and region data have simple and natural representations which allow the efficient performance of operations involving locality of reference, geometric calculations such as area computation, and set operations such as region intersection. The representation of line data, on the other hand, is more complicated. Several hierarchical structures based on quadtrees have been proposed, all with certain drawbacks, and none with the natural elegance of the adaptations representing points and lines. Our study reviews the hierarchical representation of vector data in the particular context of a geographic information system, but most of our requirements would be necessary in any application where vector data is important. A good vector representation should have the following properties. First, the data structure must represent vectors precisely rather than as digital approximations. This includes the ability to accurately represent any number of vectors intersecting at a single point. Secondly, the structure must allow the data to be updated consistently. For example, insertion and subsequent deletion of a vector should leave the data unchanged. As a more complex example, it should be possible to compute the intersection of a set of vectors with a region, and then restore the information to its original state by performing a union with the complement of the original intersection. This operation involves splitting and reassembling vector primitives. Thirdly, the structure should allow the efficient performance of primitive operations such as insertion and deletion of vector data elements, and should facilitate the performance of more complex operations such as edge following, intersection with a region, or point-in-polygon though these are somewhat application-dependent. Previous hierarchical representations for vector data have been deficient in one or more of these areas.

In this paper, we develop a data structure for the representation of vector data which has the properties described above. Section II contains a brief overview of quadtrees, while section III reviews quadtree structures for storing vector data. Section IV presents a new data structure termed a PMR quadtree and shows how it can deal with line segment fragments. Section V describes a simple implementation of the PMR quadtree while section VI reports on empirical tests. Conclusions and suggestions for future work are presented in section VII.

II. Quadtrees as Geometric/Geographic Data Structures

The quadtree [Same84b] is a hierarchical, variable resolution data structure which recursively subdivides the plane into blocks based on some decomposition rule. The technique is general and can be applied to three (octrees) and higher dimensional spaces. It may be considered as a member of a general class of hierarchical data structures based on spatial decomposition which includes k-d trees [Bent75], bintrees [Know80, Same85a], and other structures. A distinction is frequently drawn between those structures in which the subdivi-

sion boundaries are determined by the data as in the classical point quadtree [Fink74], and those in which the boundaries are pre-determined by the data structure as in the region quadtree [Klin71]. The latter is sometimes termed "regular decomposition", and the structures considered in this paper are of this type.

Because of their explicitly spatial nature, quadtrees are well suited for the representation of geometric data. The simplest example is the region quadtree where an image consisting of a set of discrete regions is represented by recursively quartering the image until every block is uniform in color. In a typical binary image, the number of blocks or leaf nodes in such a representation can be considerably less than the number of pixels in an array representation of the same image. Since many operations can be performed on a quadtree in time proportional to the number of nodes, it may be advantageous in terms of speed to manipulate data in quadtree form. Furthermore, the quadtree contains information regarding the large-scale structure of the data which is not present in a low-level representation such as an array. For point data, an analogous structure, termed the PR quadtree is formed by recursively quartering the plane until no block contains more than one data point.

We have used the above representations for areal and point data in a prior implementation of a geographic information system [Same85d]. Such simple schemes do not, however, work well for vector data. For example, attempting to divide the plane until each subdivision contains only one vector element leads to an unbounded decomposition if two vectors intersect. This reflects a basic property of lineal data. Namely, while point and area data can be adequately represented by a hierarchical decomposition of space that stores only a single piece of information per block, a similar representation of vector data requires the ability to store an arbitrary amount of data per node. Specifically, for a one item per node representation to work, the amount of information needed to describe a block must decrease as the size of that block is reduced. An intrinsic property of lineal data however, is that large amounts of information can be concentrated at a single location (e.g. when several vectors intersect at the same point). No amount of subdivision will reduce this information. Thus it is not surprising that hierarchical representation of vector data should be more difficult than point or areal data. To set our problem in a proper perspective, we review in the following, several recent proposals for the hierarchical representation of vector data that have appeared in the literature.

III. Quadtree Structures for Storing Line Data

1. The MX Quadtree

The MX quadtree [Hunt79], is probably the simplest way of representing line data, and is a region quadtree in which lines are represented by regions which are one pixel wide. It can be viewed as a quadtree representation of a chaincode. Its advantages are its relative simplicity, and the ability to represent (more or less) arbitrary space curves. Disadvantages include lack of exact representation, extreme locality of reference, large storage requirements since every point on a line is stored as a separate pixel, and lack of any structure related to the lineal nature of the data.

2. The Line Quadtree

The line quadtree [Same84a] is also based on the region quadtree, and represents curves by the boundaries of the encoded regions. This is accomplished by storing additional information about the edges of the blocks. It has the advantages of a relatively simple structure, the ability to combine region and boundary data, and is somewhat less local than the MX quadtree. The primary disadvantages are the fact that it is limited to rectilinear curves which demarcate regions, and the lack of structure based on lineal nature of data.

3. The Edge Quadtree

The edge quadtree was originally developed by Shneier [Shne81] as a method of approximating an edge in an image by recursively splitting space into quadrants until each block contains at most a single section which can be approximated by a line segment. This scheme deals only with single segments, and hence some modification is necessary to make it suitable for representing multiple intersecting lines. A variant described in Rosenfeld *et al.* [Rose83] known as the linear edge quadtree achieves this by using the decomposition rule "split until no block intersects more than one line segment or until the resolution limit is reached". Nodes containing more than one segment at the highest resolution are assigned a special type (point nodes), and a count of the number of lines intersecting the block is associated with the node. Thus point nodes indicate those places where the vectors constituting the data are too close together to be resolved. Line segments are stored in the other nodes by recording their local intersection with the edges of the block. Advantages include the ability to represent arbitrary collections of line segments, some structure based on the lineal nature of the data, and a representation that stores only one item per node (as opposed to the methods requiring the use of variable size nodes described later in this paper). Major disadvantages are the complexity of the representation and consequent difficulty of performing operations, loss of information at intersections due to the use of single a special value to label such nodes, locality of reference, and loss and degradation of information caused by separately calculating the intersection of the data segment with every block through which it passes.

4. PM quadtrees

The PM quadtree was developed by Samet and Webber [Same85c] and refers to a group of structures which store linear data in the form of line segments. The basic idea is to use some splitting rule to recursively partition the plane into quadrants, and to store with each block all the segments passing through it. This generally requires the use of variable size nodes, and for some splitting rules, the depth of decomposition generated may exceed the resolution of the segment endpoints by a considerable factor. Samet and Webber [Same85c] describe the structures resulting from three decomposition rules, which they refer to as PM1, PM2, and PM3.

The PM1 quadtree is defined by the rule "quarter until every block contains a single segment endpoint, or else it intersects just one segment, or else it is empty". The main drawback of the PM1 quadtree is that it has very bad worst case behavior in terms of the maximum depth and the number of nodes which may be generated. A one item per node variation called the segment quadtree is described by Samet [Same85b].

The PM2 quadtree is defined by the rule "quarter until every block contains a single segment, or else all segments

intersected by it have a common endpoint, or else it is empty" It has better worst-case behavior than the PM1 formulation in terms of maximum depth, but this depth is still considerably higher than the resolution of the segment endpoints.

The PM3 quadtree uses the rule "quarter until no block contains more than one endpoint". The segments passing through each block are then recorded in the node. Note that the splitting rule is just the PR rule applied to the endpoints of the line segments. The rule for the PM3 quadtree is the simplest of the three, and despite the fact that it does not refer to vectors as lineal objects, but only to their endpoints, it produces the most usable structure of the three. Figure 1 depicts a set of line segments and its PM3 quadtree. Note that the PM quadtrees essentially solve the problem of how to represent vector data exactly in a hierarchical structure. The price is the cost of implementing variable size nodes.

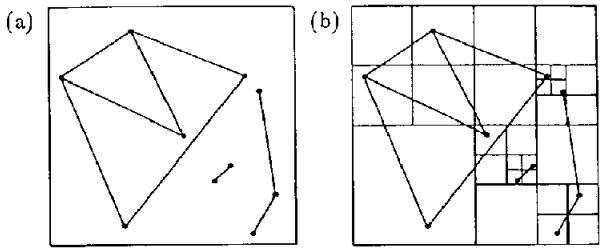

Figure 1. (a) Set of line segments and (b) corresponding PM3 quadtree.

5. Edge EXCELL

A slightly different method called *edge* EXCELL is described by Tamminen [Tamm83]. It is based on a regular decomposition that splits the cells of a grid alternately along different dimensions. A grid directory is used to map the cells into storage areas of fixed capacity (buckets) which may reside on disk. In each bucket are stored the segments that intersect the corresponding cell. When a bucket associated with a single cell overflows, every cell in the grid is split along one dimension. Overflow buckets are used to handle the case when more segments intersect at a point than can be contained in a bucket. *Edge* EXCELL is similar to the PM quadtree in that it attempts to divide space into bins containing a manageable amount of information, however it differs in that it uses fairly large bucket sizes (i.e greater than 10 data elements), while the various PM quadtrees use splitting rules which result in a low average occupancy.

IV. The PMR quadtree and fragments.

The structure that we propose uses a variant of the PM quadtree, henceforth referred to as the *PMR quadtree* as the means of controlling the amount of information stored per node. We also generalize the concept of a line segment to represent vector data in a manner that is exact and does not degrade under operations which cause a vector feature to be split or clipped. This generalization is referred to as a *fragment*.

The PMR quadtree (for PM Random) is based on the observation that any rule that divides up the line segments among quadtree blocks in a reasonably uniform fashion can be used as the basis for a PM-like quadtree. In fact, unless it is required by the application, the structure need not be uniquely determined by the data. Probabilistic splitting rules can be used as easily as any other. For instance, a rule such as "If the number of segments in a block exceeds n when a segment is added, split it once" in conjunction with a corresponding deletion rule could be used to dynamically maintain a collection of line segments.

The PMR quadtree uses a pair of rules, one for splitting and one for merging, to dynamically organize the data. The splitting rule, invoked whenever a line segment is added to a node, states "if the number of segments in the node exceeds n (four in the particular example studied,) then split the node once into quadrants". The corresponding merging rule, invoked when a segment is deleted, states "merge while the number of distinct line segments contained in the node and its siblings is less than or equal to n (four) " Figure 2 shows the construction of a PMR quadtree with the threshold n equal to two, for the segments in Figure 1. Note in figure 2b, that the insertion of segment 7 causes two blocks to split (the NW and NE quadrants) since the capacity of each of these blocks was exceeded by its insertion. Since a node is split only once when the insertion of a segment causes the threshold n to be exceeded, a node may contain more than n segments. However, except in the unusual case where many segments share an endpoint, the node occupancy is unlikely to exceed the threshold by much. This scheme differs from the structures previously considered here in that the quadtree for a given data set is not unique, but depends on the history of manipulations applied to the structure. Certain types of analysis are thus more difficult than with uniquely determined structures. On the other hand, this structure permits the decomposition of space to be based directly on the lineal data stored locally. The PMR quadtree was chosen for this reason.

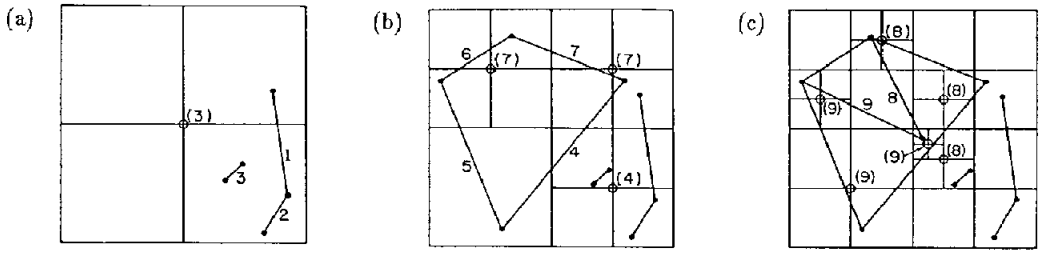

Figure 2. Building PMR quadtree from segments of Figure 1 with threshold equal to two. (a) Three segments have been inserted causing the plane to be quartered once as indicated by the small circle, (b) Segments 4-7 inserted causing three blocks to split, (c) Segments 8 and 9 inserted causing five more blocks to split.

We now address the problem of how to clip a segment in such a way that the operation may be reversed without data degradation should the missing portion be reinserted. In geographical applications, segment truncation arises when a line map is intersected with an area. Since the borders of the area may not correspond exactly with the endpoints of the segments defining the line data, certain segments may be clipped. Such an intersection is illustrated in Figure 3. The partial line segment produced is referred to as a *fragment* and the artificial endpoints produced by such an intersection are referred to as *cut points*. The problem reduces to that of representing fragments. One possible solution is to represent a fragment by introducing new, intermediate endpoints at the cut points, creating a whole new segment. In continuous space, a new segment which is colinear with the original one, but has at least one different endpoint, can be exactly represented. In discrete space (e.g., as a result of the digitization process), this is not always possible because the continuous coordinates of the cut point do not, in general, correspond exactly to any coordinates in the discrete space. If the new line segment is represented approximately in the discrete space, then the original information is degraded, and the pieces cannot reliably be rejoined. In addition, if an intermediate point is introduced to produce new segments, then the new line segment descriptor must be propagated to all remaining blocks containing the original segment. This is likely to be a very time-consuming operation.

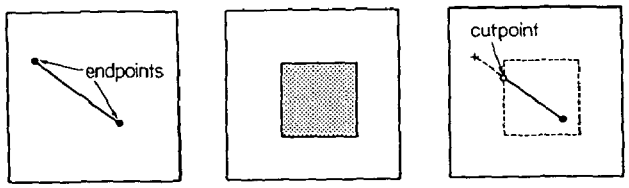

Figure 3. Definition of a fragment (c) from the intersection of a segment (a) with a region (b).

An alternative solution is to retain the description of the original segment, and use the spatial properties of the quadtree to specify what portions of the segment are actually present. The underlying insight is that a node may contain a reference to a segment, even though the entire segment is not present as a lineal feature. Rather, the segment descriptor contained in a node can be interpreted as implying the presence of just that portion of the segment which intersects the corresponding quadtree block. Such an intersection of a segment with a block will be referred to as a *q-edge*, and the original segment will be referred to as the *parent segment*. The presence or absence in the quadtree of a particular q-edge is completely independent of the presence or absence of q-edges representing other parts of the parent segment. Thus lineal features corresponding to partial segments (i.e., fragments) can be represented simply by inserting the appropriate collection of q-edges. Since the original descriptors are retained, a lineal feature can be broken into pieces and rejoined without loss or degradation of information. In the quadtree structure, q-edges are combined to represent arbitrary fragments of line segments. Since they all bear the same segment descriptor, they are easily recognizable as deriving from the same parent segment. This solves the problem of how to split a line or a map in an easily reversible manner. The use of this principle to represent the lineal feature produced by the intersection of Figure 3 is shown in Figure 4.

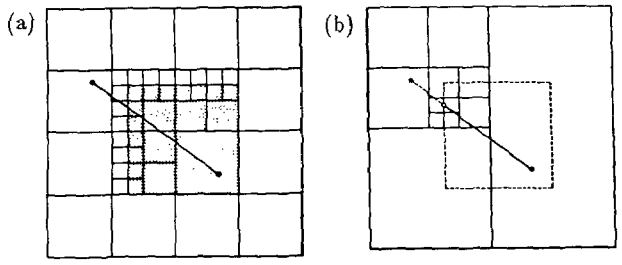

Figure 4. Representation of fragment of Figure 3 using a collection of q-edges. (a) Region quadtree for area with line segment superimposed, (b) set of five q-edges composing fragment.

The PMR quadtree can be used to represent a collection of fragments by slightly modifying the splitting and merging rules to reflect the insertion and deletion of fragments instead of line segments. The splitting rule which is invoked whenever a fragment is introduced into the structure now states "quarter until no block contains a cut point in its interior (i.e., first localize the cut points), and then once more if a block contains more than n (four) q-edges." The merge condition is now invoked both when a fragment is deleted, and when one is inserted (since a fragment may be inserted which restores the larger segment of which it is a part) and states "merge while there are n (four) or fewer distinct parent segments in the four sibling blocks and the q-edges are continuous through the block produced by the merge". Q-edges satisfying this last condition are said to be *compatible*. For example, see figure 5a where q-edges a,b and c are compatible, whereas in figure 5b, q-edges a and b are incompatible. Figure 6 shows a set of fragments produced by the intersection of the segments of Figure 1 with an area, and depicts the corresponding PMR quadtree when n is equal to two.

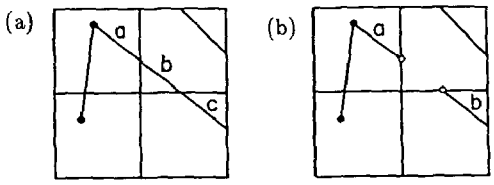

Figure 5. Sibling blocks containing (a) compatible and (b) incompatible sets of q-edges.

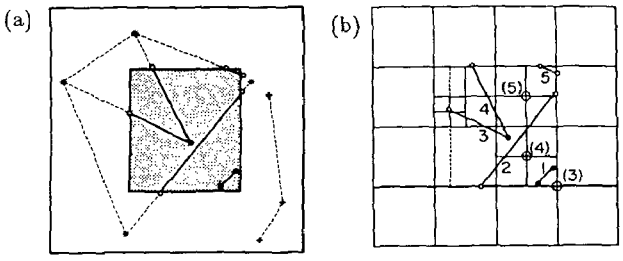

Figure 6. (a) Set of five fragments induced by the intersection of a region with the segments of Figure 1. (b) PMR-f quadtree with bucket size equal to two, for fragments of Figure 9 inserted in indicated order.The three circled splits were caused by exceeding the threshold. All others are necessary to localize the cut points of the fragments.

The above procedure is stated in terms of arbitrary fragments considered as abstract objects, but in practice some method is needed for specifying a fragment concretely. The easiest method is to restrict ourselves to the insertion and deletion of the restricted set of fragments corresponding to the intersection of line segments with all potential quadtree blocks. We will term these *q-fragments* in view of their similarity to q-edges. Arbitrary fragments can be specified to the resolution of the tree in terms of component q-fragments which thus form a set of building blocks. Note that a given fragment may be represented by different collections of q-edges in different quadtrees. The particular collection of q-edges that results from the insertion of any fragment depends upon the structure of the tree. Precise algorithms for insertion and deletion of q-fragments in PMR trees are given in the next section.

V. Implementation and algorithms.

The PMR quadtree was implemented in a geographic information system as part of an ongoing investigation into the use of of hierarchical data structures in the representation and processing of cartographic information of which only a brief description is given here. For a detailed description see [Same85d]. Since quadtrees are based on spatial decomposition, and geographical information is intrinsically spatial in nature, it was felt that quadtrees would be particularly appropriate as a basic data structure for this application. The current system includes representations and primitive operations which can be used to efficiently handle queries such as "report all wheat-growing regions within 50 miles of the Mississippi River".

Because of the large amount of information contained in geographic features, most of the data must be maintained in secondary storage. In the database system, this is achieved by use of a structure called a linear quadtree [Garg82], which is a list of the quadtree leaf nodes in the order that would be produced by a preorder traversal of the tree. The leaf nodes are represented by a pair of numbers collectively termed a *locational code*. The first number corresponds to the level of the node in the quadtree. The second is composed of a sequence of two bit directional codes that give the path from the root of the quadtree to the leaf. This process is equivalent to taking the x-y coordinate of the pixel in one corner of the leaf and interleaving the bits. The ordered list is maintained on disk as a B-tree [Come79] which is brought into core a few pages at a time. This organization enables the efficient execution of any operation that can be performed by traversing the quadtree in preorder including calculation of area, overlay, display, and connected component analysis. In fact almost any task which can be performed one scan line at a time in an array representation can be done during a single traversal in a quadtree. The B-tree structure is maintained by a kernel of primitive functions which allows the user to manipulate the structure as if it were an ordinary quadtree. The system represents areal and point data as (linear) region and PR quadtrees respectively. The similarity of these two data structures allows easy implementation of operations involving multiple data types -- for example locating all the cities with population greater than 5,000 within 20 miles of wheat-growing regions in Texas.

Since line data constitute a third major cartographic data type, a line representation is desired which is similarly compatible. The search for such a structure led us to the development of the PMR quadtree and the idea of fragments.

The first question is how to implement the variable size nodes. Since the number of line segments in a node is potentially unbounded, a true variable-length storage scheme must be used. For pointer-based quadtrees, linked lists are one possibility. An alternative is a variant of a binary tree structure that reflects the position of the segments within the block (see Samet and Webber [Same85c]). The second suggestion seems to be unnecessarily complicated for our application. One property of the PMR quadtree is that, although the maximum number of q-edges occurring in a node is potentially unbounded, the average occupancy remains low. For random vectors, it can be shown that the expected value is less than n if $n>1$. In our empirical tests, with $n=4$, the average occupancy remained less than 3 in all cases. The low average occupancy makes a linear search through the q-edges of a node practical. For linear quadtrees, ordered by the locational codes of their leaf nodes, the simplest way of implementing variable node sizes is to duplicate locational codes. This is the method used in our application. The q-edges intersecting a node are represented by a pointer to a record describing the parent segment. All q-edges with the same parent share this descriptor, which avoids unnecessary duplication of information.

We now present algorithms for the insertion and deletion of q-fragments in the PMR quadtree. At this point we should emphasize, the terminological distinction between q-edges and q-fragments. We use the term q-edge, to refer to the information content of a PMR quadtree node. Every node in the PMR quadtree contains zero or more q-edges representing the intersection of line segments with the corresponding quadtree block. We use the term q-fragment, on the other hand, to refer a member of a convenient set of primitive fragments, which just happen to be the the intersections of line segments with potential quadtree blocks. The block is conceptual in that it may not correspond to a leaf node in the quadtree into which the q-edge is being inserted. The block may correspond to a node deeper than any that exist in the tree, in which case the plane must be further subdivided. On the other hand, the block may correspond to a gray node in the quadtree in which case, several q-edges must be inserted.

We assume that a PMR quadtree is represented as a collection of pointers to records of type *node*. These conceptual nodes may contain a variable amount of information since a node may contain several q-edges. We also assume some basic routines for manipulating the structure. In the following N is a pointer to a quadtree node, L is a pointer to a record representing a line segment, and B refers to the locational code of a quadtree block. INSTALL(L,N) installs a q-edge in node N corresponding to the intersection of line segment L with the node N. REMOVE(L,N) removes q-edge corresponding to the intersection of L with N from node N (if it exists there). SPLIT(N) splits leaf node N into quadrants. MERGE(N) merges leaf node N with its siblings if they are leaf nodes. FIND(B) takes a block and returns a pointer to the corresponding node n in the tree if it exists, or to the smallest subsuming node if it does not. SPLITCOND(N) returns true iff node n contains more than (four) q-edges. MERGECOND(N) returns true iff node N is a leaf node all of whose siblings are leaf nodes, and the siblings contain (four) or fewer parent segments, and their q-edges are compatible. SON(B, direction) returns the locational code of the block which is the son of B in the given direction. SIZE(N) returns the size of a node N or block B.

Insertion of a q-fragment, say F representing the intersection of line segment L with block whose locational code is B is accomplished as follows.

(1) If B corresponds to a leaf node then F is installed, and the node is checked for splitting or merging. Merging can occur if the inserted q-edge restores a larger fragment. An example of this is shown in Figure 7.

(2) If B corresponds to a gray node, then the q-fragments corresponding to the intersection of L with the sons of B are inserted recursively. Figure 8 shows this procedure.

(3) If B is subsumed by a leaf node, say N, then N is quartered until a leaf node is produced which corresponds to B, and F is then installed as in case 1. This process is illustrated in Figure 9.

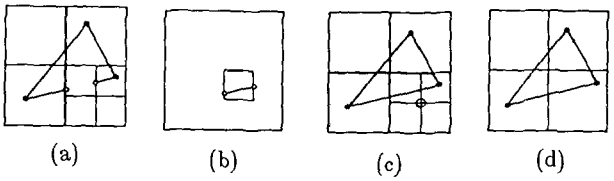

Figure 7. Insertion in PMR quadtree of a q-fragment which causes merging of blocks. (a) Quadtree before insertion, (b) q-fragment to be inserted, (c) insertion produces sibling blocks with compatible q-edges, (d) Structure after merging.

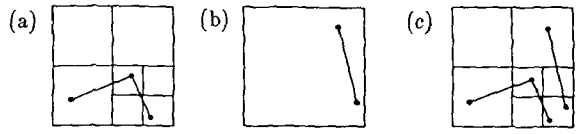

Figure 8. Insertion in PMR quadtree. Large q-fragment is inserted by decomposing it into smaller q-fragments. (a) Original quadtree, (b) q-fragment to be inserted, (c) q-fragment broken into three, smaller q-fragments whose sizes match those of extant blocks.

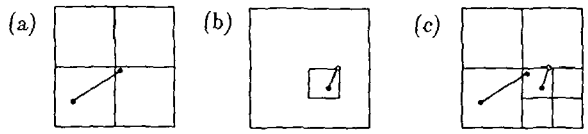

Figure 9. Insertion in PMR quadtree of small q-fragment into large block. (a) Original quadtree, (b) q-fragment to be inserted, (c) block quartered until size matches q-fragment.

More formally, we have:

```
recursive procedure INSERT(L,B);
/* Insert the q-fragment corresponding to the intersection of
   line segment L with the quadtree block whose locational
   code is B into the PMR quadtree */
begin
 value pointer line L;
 value locationcode B;
 pointer node N;
 quadrant I;
 N ← FIND(B);
 if SIZE(N) = SIZE(B) then
  begin
   INSTALL(L,N);
   if SPLITCOND(N) then SPLIT(N)
   else if MERGECOND(N) then MERGE(N);
  end
 else if SIZE(N) < SIZE(B) then
  begin
   for I in {'NW', 'NE', 'SW', 'SE'}
    do INSERT(L, SON(B,I));
  end
 else
  begin
   while SIZE(N) > SIZE(B) do
    begin
     SPLIT(N);
     N ← FIND(B);
    end;
   INSTALL(L,N);
   if SPLITCOND(N) then SPLIT(N)
   else if MERGECOND(N) then MERGE(N);
  end;
end;
```

Deletion of a q-fragment, say F, representing the intersection of a line segment L with a block whose locational code is B, is accomplished by a similar procedure. Deletion of a q-fragment is interpreted as erasing a portion of the parent segment.

(1) If B corresponds to a leaf node which contains F, then the reference to F is removed and a check for merging is made on the node and its siblings.

(2) If B corresponds to a gray node, then the q-edges formed by intersecting L with the sons of B are deleted recursively.

(3) If B is subsumed by a leaf node, then that node is quartered until a leaf node is produced which corresponds to B, and F is deleted as in 1.

More formally, we have

```
recursive procedure DELETE(L,B);
/* Delete the q-fragment corresponding to the intersection of
   line segment L with the quadtree block whose locational
   code is B from the PMR quadtree */
begin
 value pointer line L;
 value locationcode B;
 pointer node N;
 quadrant I;
 N ← FIND(B);
 if SIZE(N) = SIZE(B) then
  begin
   REMOVE(L,N);
   if MERGECOND(N) then MERGE(N);
  end
 else if SIZE(N) < SIZE(B) then
  begin
   for I in {'NW', 'NE', 'SW', 'SE'}
    do DELETE(L, SON(B,I));
  end
 else
  begin
```

```
    while SIZE(N) > SIZE(B) do
     begin
      SPLIT(N);
      N ← FIND(B);
     end;
    REMOVE(L,N);
    if MERGECOND(N) then MERGE(N);
   end;
end;
```

VI. Tests and Comparisons

In order to evaluate the performance of the proposed line representations, tests were run using geographic data on four different structures: MX, edge, PM3, and PMR quadtrees. PM3 quadtrees were implemented using fragments in a manner analogous to our PMR implementation. The first two, as discussed in earlier sections of this paper, have deficiencies that ultimately make them unsuitable for the desired application. However, enough can be done with them to allow a meaningful comparison to be made for some operations. In particular, if the performance of the PM methods is far worse than that of methods known to be deficient other ways, we must ask whether the gain is worth the cost. The following tests were made.

(1) Time required to build the quadtree structure.
(2) Comparison of the storage requirements of the different representations.
(3) Time required to perform an intersection with an area.
(4) Comparison of the effect of different values for the split threshold n for PMR quadtrees.

Three different lineal data sets were used in the tests. The data are in the form of connected line segments and correspond to maps of three different geographic features: a railroad line, a city boundary, and a road map for the city. The first is very simple and contain only 16 segments. The city boundary is a simple closed curve containing 64 segments. The road map is fairly complex and contains 764 segments (figure 10). The vector endpoints were digitized onto a 512 x 512 grid which is the size of the space used for the experiments.

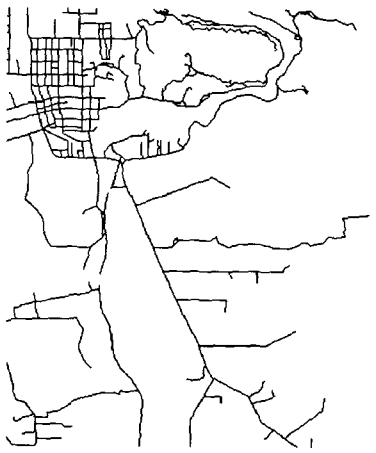

Figure 10. Set of 764 line segments consituting road map.

1. Building test.

The building algorithm essentially tests the efficiency of insertion into the structure. The three maps were built for each of the four structures. Results are displayed in Table 1. Utime refers to the actual runtime of the algorithm. The data indicate that none of the methods is overwhelmingly superior in terms of insertion efficiency, but the PMR representation has a definite if irregular lead in most cases. In the case of the road map, which represents the most realistic data set, the MX, edge, and PMR representations are more or less equivalent, and about 30% faster than the PM3.

2. Tree sizes.

The final size of the structure is important because it is the form in which the information is stored within the system. To a lesser extent, the size is important because many of the algorithms run in time proportional to the number of nodes (i.e., the size) of the quadtree. However, the constants of proportionality may differ between different representations, so a comparison in this respect is not very meaningful without additional information. The results are tabulated in Table 1 for the three maps and methods. The term "leaves" refers to the number of quadtree leaf nodes for the MX and edge quadtrees, and to the sum of the number of q-edges and the number of empty nodes for the PM quadtrees. A quadtree leaf and a q-edge pointer occupy the same amount of storage, so the numbers represent comparable quantities. The term "qnodes" refers to the number of nodes in the PM methods and is included to provide a feeling for the fullness of the nodes and to support the claim that the average occupancy is indeed small (less than three for all examples here), for realistic geographic data sets.

Table 1: Building times and sizes

Map	Structure	Utime	Leafs	qnodes
railroad	MX	2.21	2101	-----
	edge	.63	301	-----
	PM3	.63	92	70
	PMR	.30	35	19
city	MX	2.62	2347	-----
	edge	2.25	835	-----
	PM3	2.36	310	214
	PMR	1.28	151	70
road	MX	22.55	19699	-----
	edge	20.48	7723	-----
	PM3	29.38	3939	2350
	PMR	19.03	2078	874

Examination of the results reveals a steady decrease in the required storage from MX to edge to PM3 to PMR. The PMR representation is at least eight times more efficient in its use of storage than the simple MX in all the cases tested, but both PM techniques improve the storage efficiency significantly over the other two techniques. This improvement can be explained by noting that the PM quadtrees use one-dimensional primitives which can extend over distances of many pixels rather than the pixel by pixel representation that is used by the MX method exclusively, and by the edge method when segments approached each other. It should be mentioned that these results are for collections of complete line segments since no cut-points were involved in the original maps. The presence of cut-points in the data would be expected to reduce the storage efficiency, since further decomposition would be required to localize them.

3. Intersection test.

The intersection function is a high level geographic computation which involves processing the entire data structure. In the case of the PM quadtrees, it tests the efficiency of the fragment representation, because the previously intact line segments are now cut where they cross an area boundary. Because the PM/fragment methods enable the performance of operations not possible with the local methods (e.g., reassembling split lines without degradation of data) it is not entirely clear that the different intersection computations are comparable: however, the results may give a general idea of the practicality of high level operations. The intersections were performed using the roadmap as the lineal data set (the others being too small to provide a reasonable intersection,) and three binary maps and their complements, represented in the form of area quadtrees, as templates. The use of the complements is intended to permit the effects of the size and shape of the templates to be distinguished from the overall efficiency of the different algorithms. This precaution is necessary because the intersection algorithm used with the PM structures works differently than the one used with the two other methods, and is affected differently by changing the shape of the template. In particular, the intersection procedure for the MX and edge quadtrees works by inserting into a blank map all linear sections that intersect the template, while the procedure for the PM quadtrees works by erasing the sections of the line map which do not intersect the template. It turns out that insertion and deletion of q-edges are operations of comparable complexity. For a map produced by erasing portions of a preexisting map, the number of deletions corresponds to the number of insertions necessary to produce the complementary map since the same q-edges are involved. Hence it is more appropriate to compare the intersections of the first two representations with the complementary intersections of the PM/fragment representations.

The three templates used are referred to as center, stone, and pebble, and represent a floodplain in register with the road map, and unrelated binary images derived from thresholded photographs of stones and pebbles respectively. Only the floodplain map has any geographic relevance. The other two were used with the intention of giving the system a more stringent, if less realistic, test. In particular, the degree of fragmentation induced by the pebble map probably exceeds any that would normally arise in a geographic application.

The results of the tests are given in raw form in Table 2, and are apparently ambiguous. In some cases the PM methods take much longer than the MX and edge schemes, but in others they take less time (though not correspondingly so). This inconsistency is due to the complementary effect of the different intersection algorithms discussed above. Table 3 reorganizes the data so that the appropriate complements are compared, and a consistent trend is now apparent. The time needed to perform an intersection generally increases from MX to edge to PM3 to PMR with the PM methods taking somewhere around twice as long as their competitors. Note that the order in which the intersection times increase is the same in which the structure sizes decrease suggesting that we are observing a time versus space trade-off.

Table 2 also gives the sizes of the structures representing the intersections. Comparing the sizes of the resulting maps reveals that, as predicted above, the improvement in storage requirements from MX to PMR is less dramatic than when no cut points were present. The degree of fragmentation varies, but in the case of intersections with the pebble map and its complement, it is probable that few if any of the original segments are intact. Since additional splitting is required to localize the fragment ends, the representation is not as efficient as for data that contains no cut points. The improvement is still present, however, with the PM methods generally requiring between one half and one quarter of the space of the MX, and significantly less than the edge quadtree. The order of decreasing sizes from MX to PMR which was noted for the segment data remains the same.

Table 2: Intersection times and sizes

Intersection	Structure	Utime	Leafs	qnodes
road & center	MX	4.10	3094	-----
	edge	5.60	1759	-----
	PM3	15.50	1019	874
	PMR	14.60	910	775
road & centercomp	MX	16.90	17314	-----
	edge	14.50	8320	-----
	PM3	6.83	4275	2677
	PMR	8.02	2568	1402
road & stone	MX	4.90	3397	-----
	edge	8.70	2344	-----
	PM3	22.40	2011	1774
	PMR	28.10	1853	1651
road & stonecomp	MX	19.70	17776	-----
	edge	19.60	8803	-----
	PM3	15.30	4684	3244
	PMR	22.00	3270	2122
road & pebble	MX	11.45	9022	-----
	edge	17.25	5653	-----
	PM3	32.20	4530	3760
	PMR	47.00	4034	3370
road & pebblecomp	MX	16.95	13564	-----
	edge	20.80	7459	-----
	PM3	30.20	5086	4078
	PMR	42.00	4250	3436

Table 3: Reordered intersection times

Intersection	Structure	Utime
road & center	MX	4.10
	edge	5.60
road & centercomp	PM3	6.83
	PMR	8.02
road & centercomp	MX	16.90
	edge	14.50
road & center	PM3	15.50
	PMR	14.60
road & stone	MX	4.90
	edge	8.70
road & stonecomp	PM3	15.30
	PMR	22.00
road & stonecomp	MX	19.70
	edge	19.60
road & stone	PM3	22.40
	PMR	28.10
road & pebble	MX	11.45
	edge	17.25
road & pebblecomp	PM3	30.20
	PMR	42.00
road & pebblecomp	MX	16.95
	edge	20.80
road & pebble	PM3	32.20
	PMR	47.00

At first glance, the results seem disappointing because we have come to expect, since so many quadtree algorithms can be made to run in time proportional to the number of nodes, that a decrease in the size of a structure will imply a corresponding decrease in execution time for operations performed using that structure. There is however, no reason to expect this property to hold across different structures, since the amount of work done per node will certainly differ. On the other hand, the increased execution time is by no means severe enough to damage the value of the representation. This is especially true in light of the fact that the PM/fragment structures have capabilities and a certain elegance that the MX and edge quadtrees completely lack. This is worth a certain price.

4. Different splitting thresholds.

In our implementation of the PMR quadtree, we chose n=4 as the threshold at which to split a node. It was clear from the start that both very low and very high thresholds would degrade the performance of the structure. For low splitting thresholds, say one or two, the storage requirements would tend to be high, since a large amount of splitting would take place in a futile attempt to separate intersecting vector features. Conversely, a high average occupancy would unduly increase the amount of effort involved in processing each node, the extreme example being a single node containing a list of all the vector features, which would completely nullify any benefits obtained from the spatial decomposition. We initially selected a threshold of four because that was the greatest number of roads likely to intersect at a single point. Tests were run for thresholds of 1,2,4,8,16 and 32 on two data sets: the road map, and a collection of 100 randomly intersecting line segments. The segments for the road map form a planar graph, but no such restriction was imposed upon the random segments of the second data set.

The times and sizes for the building and intersection algorithms are given in tables 4 and 5. For the roadmap, the results are as expected. There is a slow decrease in building time, and a rapid decrease in storage requirements as the threshold increases. This is not surprising since generally less decomposition is being done. For the intersection algorithm, the time increases for both high and low thresholds as predicted, with a minimum at n=4. The storage requirements of the PMR quadtree representing the intersection decrease to an asymptotic value as the threshold increases. This is the point at which all decomposition is due to the localization of fragment endpoints, and further changes in the threshold consequently have no effect.

The results for the random segments are very similar. The optimal threshold for the intersection algorithm is 8 instead of 4, but the difference is very small. The somewhat greater sensitivity of the structure storage requirements to the threshold is due to the fact that the road map contains many very short segments with the result that there are fewer opportunities for merging when the threshold is increased. The fact that the performance of PMR quadtree is similar for radically different types of vector data, suggests that it would provide a robust, general vector representation.

Table 4a: Results of building PMR road map using different thresholds.

Threshold	Utime	Leaves	Qnodes
1	26.9	5765	3745
2	22.5	4082	2404
4	16.6	2078	874
8	14.7	1396	346
16	13.7	1145	163
32	15.6	1001	73

Table 4b: Intersection of PMR road map with center using different thresholds.

Threshold	Utime	Leaves	Qnodes
1	14.7	1069	925
2	13.5	968	829
4	12.0	910	775
8	13.8	897	763
16	17.8	895	760
32	22.0	895	760

Table 5a: Results of inserting 100 random segments using different thresholds

Threshold	Utime	Leaves	Qnodes
1	31.1	9189	6508
2	20.4	5970	3433
4	11.6	3031	1114
8	6.3	1407	262
16	4.2	763	79
32	3.4	452	25

Table 5b: Intersection of 100 random segments with center using different thresholds.

Threshold	Utime	Leaves	Qnodes
1	23.5	2496	2131
2	17.9	2159	1804
4	16.1	1880	1582
8	15.8	1807	1540
16	19.3	1774	1531
32	27.2	1774	1531

VII. Conclusions

The PMR quadtree used to store segment fragments provides a hierarchical vector representation that satisfies the conditions set forth at the beginning of this paper. It is exact (i.e., non digitized). It allows consistent updating of the data, and permits primitive vector features to be manipulated as well as cut or clipped and reconstructed without data degradation. It facilitates efficient insertion and deletion of vector elements, and the performance of high level operations such as area intersection. Since the endpoints of the vectors are not used to direct the spatial decomposition, both planar and non-planar graphs can be efficiently represented.

The basic technique, is not however, limited to representing lineal data in two dimensions. The idea of using probabilistic splitting and merging rules to recursively decompose the space and dynamically organize the data into bins of manageable average size is a powerful general notion. The PMR quadtree generalizes immediately to the representation of

lines or faces in three or more dimensions. For some recent work involving PM quadtrees in 3 dimensions, see [Ayal85, Carl85, Fuji85]. Moreover, the object represented by the q-edge need not be a line segment, but could be any entity for which there exists a descriptor, and which has definite spatial extent. The representation of lineal data could be extended for example, by having "generalized q-edges" represent the intersection of the blocks with cubic splines rather than simply with line segments. As another example, consider an image processing environment where a generalized q-edge could represent the presence or absence of a proposed object in the block. Several hypotheses could be maintained simultaneously, and updated as processing progressed. As a third example, the generalized q-edge could represent the intersection of a rectangular region with the block in a system for VLSI design rule checking.

To sum up, the PM quadtrees using fragments provide a conceptually clean representation for lineal data which explicitly addresses its one-dimensional character. For the geographic information system, the structures compare favorably in performance with the cruder MX and edge quadtrees in cases where they can be compared, and resolve the problems of loss of information and degradation of data which encumbered the latter. Finally, the structures have the potential for use as a general representation in any application where the spatial relationship of objects is important.

Acknowledgments This work was supported in part by the National Science Foundation under Grant DCR–83–02118. We have benefited greatly from discussions with Clifford A. Shaffer and Robert E. Webber.

References

1. [Ayal85] - D. Ayala, P. Brunet, R. Juan, and I. Navazo, Object representation by means of nonminimal division quadtrees and octrees, *ACM Transactions on Graphics 4*, 1(January 1985), 41-59.

2. [Bent75] - J.L. Bentley, A survey of techniques for fixed radius near neighbor searching, SLAC Report No. 186, Stanford Linear Accelerator Center, Stanford University, Stanford, CA, August 1975.

3. [Carl85] - I. Carlbom, I. Chakravarty, and D. Vanderschel, A hierarchical data structure for representing the spatial decomposition of 3-D objects, *IEEE Computer Graphics and Applications 5*, 4(April 1985), 24-31.

4. [Come79] - D. Comer, The Ubiquitous B-tree, *ACM Computing Surveys 11*, 2(June 1979), 121-137.

5. [Fink74] - R.A. Finkel and J.L. Bentley, Quad trees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1(1974), 1-9.

6. [Fuji85] - K. Fujimura and T.L. Kunii, A hierarchical space indexing method, *Proceedings of Computer Graphics'85*, Tokyo, 1985, T1-4, 1-14.

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

8. [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.

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

10. [Rose83] - A. Rosenfeld, H. Samet, C. Shaffer, and R.E. Webber, Application of hierarchical data structures to geographical information systems phase II, Computer Science TR 1327, University of Maryland, College Park, MD, September 1983.

11. [Same84a] - H. Samet and R.E. Webber, On encoding boundaries with quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 3(May 1984), 365-369.

12. [Same84b] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

13. [Same85a] - H. Samet and M. Tamminen, Efficient component labeling of images of arbitrary dimension, Computer Science TR-1480, University of Maryland, College Park, MD, February 1985.

14. [Same85b] - H. Samet, C.A. Shaffer, and R.E. Webber, The segment quadtree: a linear quadtree-based representation for linear features, *Proceedings of Computer Vision and Pattern Recognition 85*, San Francisco, June 1985, 385-389.

15. [Same85c] - H. Samet and R.E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222.

16. [Same85d] - H. Samet, A. Rosenfeld, C.A. Shaffer, R.C. Nelson, Y-G. Huang, and K. Fujimura, Application of hierarchical data structures to geographic information systems: phase IV, Computer Science TR-1578, University of Maryland, College Park, MD, December 1985.

17. [Shne81] - M. Shneier, Calculations of geometric properties using quadtrees, *Computer Graphics and Image Processing 16*, 3(July 1981), 296-302.

18. [Tamm83] - M. Tamminen, Performance analysis of cell based geometric file organizations, *Computer Vision, Graphics, and Image Processing 24*, 2(November 1983), 168-181.