EFFICIENT PROCESSING OF SPATIAL QUERIES IN LINE SEGMENT DATABASES*

Erik G. Hoel†
Statistical Research Division
Bureau of the Census
Washington, DC 20233

Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Abstract

A study is performed of the issues arising in the efficient processing of spatial queries in large spatial databases. The domain is restricted to line segment databases such as those found in transportation networks and polygonal maps. Three classes of queries are identified. Those that deal with the line segments themselves, those that involve both the line segments and the space from which they are drawn (e.g., proximity queries), and those that involve attributes of the line segments. Handling the three types of queries requires that the line segments be stored implicitly using a bucketing approach on the space from which they are drawn. A number of bucketing approaches are examined and the PMR quadtree is chosen as the most suitable representation. Its storage and execution time requirements are evaluated in the context of finding the nearest line segment to a given point. This operation is shown to take time proportional to the splitting threshold (similar to the bucket capacity) and is independent of the density of the data. The evaluation uses the road networks in the data of the U.S. Bureau of the Census.

Keywords and phrases: large spatial databases, spatial queries, spatial access methods, bucketing methods, lines, spatial indexing, spatial data structures, hierarchical data structures, geographic information systems, PMR quadtrees

---

*This work was supported in part by the Bureau of the Census under Joint Statistical Agreement 88-21 and the National Science Foundation under Grant IRI-90-17393.

†Also with the Center for Automation Research at the University of Maryland.

1 INTRODUCTION

Spatial data consists of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension which includes time (e.g., [Same90a, Same90b]). Spatial databases permit the storage of spatial information about objects (e.g., [Buch90]). In many standard database applications it is useful to add spatial attributes to describe different objects in the database such as the extent of a given river (e.g., does the Missouri River pass through the state of Missouri?), what is the boundary of St.Mary's county?, etc. In general, spatial information can be stored either explicitly or implicitly.

The conventional approach to dealing with spatial data is to store it explicitly. This is usually quite easy to do since a database management system is just a collection of records, where each record has many fields. In particular, we simply add a field to the record that deals with the desired item of spatial information. This approach is fine if we know a priori the type of spatial information that we wish to extract from our database. Unfortunately, this is not often the case, as usually we cannot predict the nature of the user's query.

In contrast, the implicit approach stores the spatial data in a way that enables it to be used to respond to the queries. In such a case, the issue of representation becomes more important since its utility depends to a large extent on the nature of the queries. In this paper we concentrate on the implicit approach.

We focus on a database consisting of a large collection of line segments such as that used in the Bureau of the Census TIGER/Line file [Bure89] for representing the roads and other geographic features in the US. The underlying representation of the data stored in such a database depends on the nature of the queries that are expected to be posed.

This paper is organized as follows. We first study the issues that must be considered in choosing a representation for a large collection of line segments. This depends on the nature of the queries involving them, and on the type of spatial operations that must be performed to answer them. The queries must include the ability to find nearest line segments (i.e., proximity queries) as well as access their attributes. This requires a data structure that sorts the line segments and we select the PMR quadtree as our representation. The rest of the paper shows how the PMR quadtree can be used to respond to our spatial queries and evaluates its performance (in terms of storage and execution time) in dealing with the TIGER/Line files. The evaluation is in terms of a query that takes a point and finds its nearest line segment. Conclusions are drawn with respect to possible improvements, and avenues for future research are suggested.

2 SPATIAL INDEXING

Each record in a database management system can be conceptualized as a point in a multidimensional space. This analogy is used by many researchers (e.g., [Hinr83, Oren84, Jaga90]) to deal with spatial data as well by use of suitable transformations that map the spatial object into a point (termed a *representative point*) in either the same (e.g.,

[Jaga90]), lower (e.g., [Oren84]), or higher (e.g., [Hinr83]) dimensional spaces.

Unfortunately, this analogy is not always appropriate for spatial data. One problem is that the dimensionality of the representative point may be too high [Oren89]. One solution is to approximate the spatial object by reducing the dimensionality of the representative point. Another more serious problem, and the one we focus on in this paper, is that use of these transformations does not preserve proximity. It is our belief that spatial data must be sorted. We are not unique in this view. However, what separates our work from that of others is that we also take the extent of a spatial object into account.

To see the drawback of just mapping spatial data into points in another space, consider the representation of a database of line segments (e.g., [Jaga90]). We use the term *polygonal map* to refer to such a line segment database, consisting of vertices and edges, regardless of whether or not the line segments are connected to each other. Such a database can arise in a network of roads, power lines, rail lines, etc. Using a representative point, each line segment can be represented by its endpoints[1]. This means that each line segment is represented by a tuple of four items (i.e., a pair of x coordinates and a pair of y coordinates). Thus, in effect, we have constructed a mapping from a two-dimensional space (i.e., the space from which the lines are drawn) to a four-dimensional space (i.e., the space containing the representative point corresponding to the line).

This mapping is fine for storage purposes. However, it is not ideal for spatial operations involving search. For example, suppose we want to detect if two lines are near each other, or, alternatively, to find the nearest line to a given point or line. This is difficult to do in the four-dimensional space since proximity in the two-dimensional space from which the lines are drawn is not necessarily preserved in the four-dimensional space into which the lines are mapped. In other words, although the two lines may be very close to each other, the Euclidean distance between their representative points may be quite large.

Thus we need different representations for spatial data. We believe that data structures based on spatial occupancy provide the best solution to these problems. Spatial occupancy methods are based on the decomposition of the space from which the data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets*. Spatial occupancy methods are also known as bucketing methods. Traditionally, bucketing methods such as the grid file [Niev84], BANG file [Free87], LSD trees [Henr89], buddy trees [Seeg90], etc. have always been applied to the transformed data. In contrast, we are interested in bucketing methods that are applied to the space from which the data is drawn (i.e., two-dimensions in the case of a collection of line segments). Moreover, our interest is in bucketing methods that are designed specifically for the spatial data type that is being stored (e.g., a collection line segments), whereas the traditional approach is to tailor the transformation to the spatial data type.

There are four principal approaches to decomposing the space from which the data is drawn. One approach buckets the data based on the concept of a minimum bounding (or enclosing) rectangle. In this case, objects are grouped (hopefully by proximity) into

[1]Of course, there are other representations as well, but they suffer from similar problems. We shall use this example in the rest of our discussion.

hierarchies, and then stored in another structure such as a B-tree [Come79]. The R-tree (e.g., [Gutt84]) is an example of this approach. The drawback of these methods is that they do not result in a disjoint decomposition of space. The problem is that an object is only associated with one bounding rectangle, yet the area that it spans may be included in several bounding rectangles. This means that when we wish to determine which object is associated with a particular point in the two-dimensional space from which the objects are drawn, we may have to search the entire database.

The other approaches are based on a decomposition of space into disjoint cells, which are mapped into buckets. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules and by the way in which the cells are aggregated. The price paid for the non-disjointness is that in order to determine the area covered by a particular object, we have to retrieve all the cells that it occupies.

The first method based on disjointness partitions the objects into arbitrary disjoint subobjects and then groups the subobjects in another structure such as a B-tree. The partition and the subsequent groupings are such that the bounding rectangles are disjoint at each level of the structure. The R$^+$-tree [Ston86, Falo87] and the cell tree [Günt87] are examples of this approach. They differ in the data with which they deal. The R$^+$-tree deals with collections of rectangles while the cell tree deals with convex polyhedra.

Methods such as the R$^+$-tree and the cell tree have the drawback that the decomposition is data-dependent. This means that it is difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations). In contrast, the remaining two methods, while also yielding a disjoint decomposition, have a greater degree of data-independence. They are based on a regular decomposition. We can either decompose the space into blocks of uniform size (e.g., the uniform grid [Fran84]) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach [Tamm82, Same85, Nels86, Nels87, Oren89]). In the former case, all the blocks are of the same size. In the latter case, the widths of the blocks are restricted to be powers of two, and their positions are also restricted.

The uniform grid is ideal for uniformly distributed data, while quadtree-based approaches are suited for arbitrarily distributed data. In the case of uniformly distributed data, quadtree-based approaches degenerate to a uniform grid, albeit they have a higher overhead. Both the uniform grid and the quadtree-based approaches lend themselves to set-theoretic operations and thus they are ideal for tasks which require the composition of different operations and data sets. In general, since spatial data is not usually uniformly distributed, the quadtree-based approaches seem to be the most flexible and therefore are the ones that we focus on in the rest of this paper.

These methods are characterized as employing spatial indexing because with each block we only store information with respect to whether or not it is occupied by the object or part of the object. This information is usually in the form of a pointer to a descriptor of the object. For example, in the case of a collection of line segments in the uniform grid of Figure 1, the shaded block only records the fact that a line segment crosses

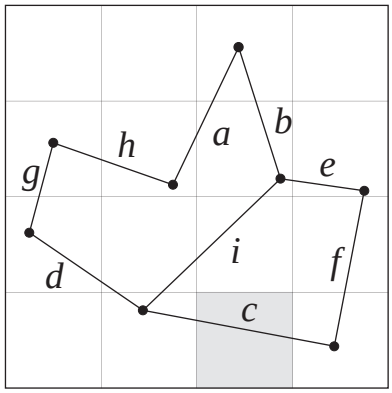

Figure 1: Uniform grid for a collection of line segments.

it or passes through it. The part of the line segment that passes through the block (or terminates within it) is termed a *q-edge*. Each q-edge in the block is represented by a pointer to a record containing the endpoints of the line segment of which the q-edge is a part [Nels86]. This pointer is really nothing more than a spatial index and hence the use of this term to characterize this approach. Thus no information is associated with the shaded block as to what part of the line (i.e., q-edge) crosses it. This information can be obtained by clipping [Fole90] the original line segment to the block. This is important for often we do not have the necessary precision to compute these intersection points anyway.

3 QUERIES ON LINE SEGMENT DATABASES

Queries on line segment databases fall into three classes. The first class consists of queries about the line segments themselves. With the exception of the points that lie on the line segments, these queries do not involve any points in the space from which the line segments are drawn (i.e., the space that they occupy). Some examples of the first class include [Jaga90]:

1. Find all the line segments that intersect a given point or set of points.
2. Find all the line segments that have a given set of endpoints.
3. Find all the line segments that intersect a given line segment.
4. Find all the line segments that are coincident with a given line segment.

Answering queries in the first class only requires that we have knowledge about the line segments themselves. Thus representation techniques that transform the line segments into points in another space are often adequate to answer them. For example, each line segment can be represented by a point in a four-dimensional space consisting of the values of the x and y coordinates of its endpoints. It can also be represented as a point in a two-dimensional space consisting of its slope and appropriate intercept value. A variant of this approach is taken by Jagadish [Jaga90] in conjunction with an LSD tree [Henr89].

The second class is broader than the first class in that it involves both the line segments and all points in the space from which the line segments are drawn. This means that proximity queries are allowed. Some examples of the second class include:

1. Find the nearest line segment to a given point.
2. Find all the line segments within a given distance from a given point (also known as a window query or a range query).

Answering queries in the second class is greatly facilitated when the line segments are sorted. Representation techniques that transform the line segments into points in another space are inadequate to answer them since they do not preserve the proximity in the space from which the line segments are drawn. However, the bucketing methods are fine for these queries.

The third class consists of queries that involve attributes of the line segments. In particular, once we have a database of line segments, it is natural to associate a type with them (e.g., road, railway line, power line, telephone line, river, etc.). The line segments can also be aggregated into higher level units such as roads, transportation networks, polygons, etc. For example, consider a decomposition of a state map into counties where each county consists of one or more polygons. Attributes give rise to more complex queries which involve more than just finding the nearest neighbor. Given our initial assumption that spatial data is stored implicitly means that we must also have the ability to extract polygons given a line segment or a point. Some examples of the third class include:

1. Given a point, find the closest line segment of a particular type. An additional optional argument can indicate a maximum distance so as to constrain the search.
2. Given a point, find the minimum enclosing polygon whose constituent line segments are all of a specified type.
3. Given a point, find all the polygons that are incident on it.

These queries have much applicability. For example, query 1 can be used with school district boundaries. In this case, once we have located the nearest school boundary it is a simple matter to find the location of the nearest school. Moreover, we assume that the identity of the two adjacent schools is stored with each boundary segment. As another example, query 2 can be used to determine the extent of two-dimensional regions by just giving a point within them. Query 1 can be varied by asking for a pruned polygon — i.e., one in which all line segments that lie inside the polygon are removed, such as a cul de sac.

The bucketing approach coupled with a procedure to extract a polygon given a line segment is useful in answering queries in the third class. Extracting a polygon involves locating an edge and the side associated with the desired region. Once this is done, we simply access one of the endpoints, examine the incident edges, and then identify the appropriate next edge to follow. In the rest of this paper we study the use and performance of the PMR quadtree, an adaptive bucketing method [Nels86, Nels87], in answering queries in the second and third classes.

4 PMR QUADTREES

The simplest representation of line data such as that comprising a polygonal map is in the form of vectors which are usually specified in the form of lists of pairs of x and y coordinate values corresponding to their start and endpoints. The vectors are usually ordered by their connectivity. Given a random point in space, it is very difficult to find the nearest line to it using such a representation. The problem is that the lines are not sorted. Nevertheless, the vector representation is used in many commercial systems (e.g., ARC/INFO [Peuq90]) on account of its compactness.

In contrast, we adaptively sort the line segments into buckets of varying size. There is a one-to-one correspondence between buckets and blocks in the two-dimensional space from which the line segments are drawn. There are a number of approaches to this problem [Same90a]. They differ by being either vertex based or edge based. Their implementations make use of the same basic data structure. All are built by applying the same principle of repeatedly breaking up the collection of vertices and edges (making up the polygonal map) into groups of four blocks of equal size (termed *brothers*) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure. This is achieved by successively weakening the definition of what constitutes a legal block, thereby enabling more information to be stored in each bucket.

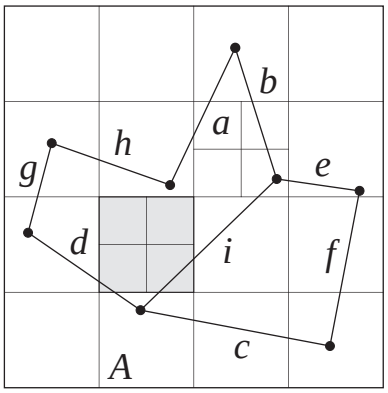

Figure 2: PM_1 quadtree for the collection of line segments of Figure 1.

The PM quadtrees of Samet and Webber [Same85] are vertex-based. We illustrate the PM_1 quadtree. It is based on a decomposition rule stipulating that partitioning occurs as long as a block contains more than one line segment unless the line segments are all incident at the same vertex which is also in the same block (e.g., Figure 2). A similar representation has been devised for three-dimensional polyhedral data, where the decomposition criteria are such that no block contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge (see [Same90a] for more details).

The PMR quadtree [Nels86, Nels87] is an edge-based variant of the PM quadtree (see also edge-EXCELL [Tamm81]). It makes use of a probabilistic splitting rule. A block is permitted to contain a variable number of line segments. The PMR quadtree is constructed

by inserting them one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all of the blocks that it intersects or occupies in its entirety. During this process, the occupancy of each affected block is checked to see if the insertion causes it to exceed a predetermined *splitting threshold*. If the splitting threshold is exceeded, then the block is split *once*, and only once, into four blocks of equal size. The rationale is to avoid splitting a node many times when there are a few very close lines in a block. In this manner, we avoid pathologically bad cases. For more details, see [Nels86].

A line segment is deleted from a PMR quadtree by removing it from all the blocks that it intersects or occupies in its entirety. During this process, the occupancy of the block and its siblings (the ones that were created when its predecessor was split) is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the block and its siblings, then they are merged and the merging process is recursively reapplied to the resulting block and its siblings. Notice the asymmetry between the splitting and merging rules.

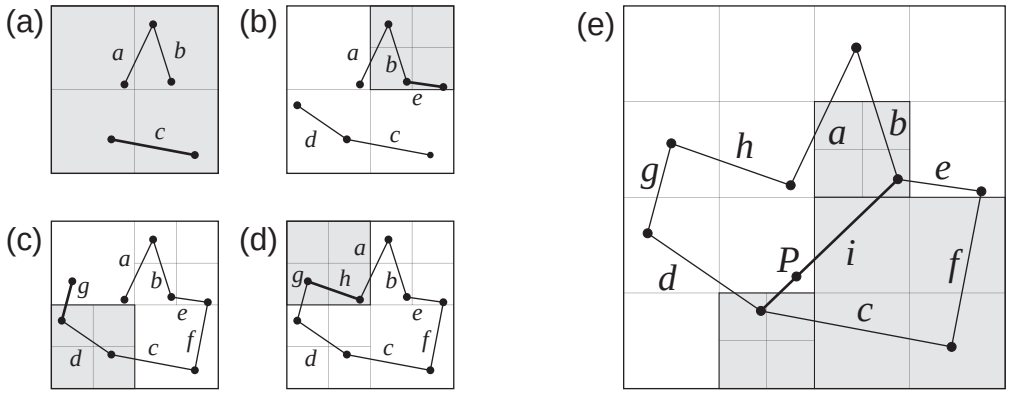

Figure 3: PMR quadtree for the collection of line segments of Figure 1. (a) - (e) illustrate snapshots of the construction process with the final PMR quadtree given in (e).

Figure 3(e) is an example of a PMR quadtree corresponding to a set of 9 edges labeled a–i inserted in increasing order. Observe that the shape of the PMR quadtree for a given polygonal map is not unique; instead it depends on the order in which the lines are inserted into it. In contrast, the shape of the PM_1 quadtree is unique. Figure 3(a)–(e) shows some of the steps in the process of building the PMR quadtree of Figure 3(e). This structure assumes that the splitting threshold value is two. In each part of Figure 3(a)–(e), the line segment that caused the subdivision is denoted by a thick line, while the gray regions indicate the blocks where a subdivision has taken place. The insertion of line segments c, e, g, h, and i cause the subdivisions in parts a, b, c, d, and e, respectively, of Figure 3. The insertion of line segment i causes three blocks to be subdivided (i.e., the SE block in the SW quadrant, the SE quadrant, and the SW block in the NE quadrant). The final result is shown in Figure 3(e). Note the difference from the PM_1 quadtree in Figure 2 — i.e., the block containing point P in Figure 3(e) is decomposed in the PM_1 quadtree while the NE block of the SW quadrant is not decomposed in the PMR quadtree.

We prefer, and use, the PMR quadtree as it results in far fewer subdivisions than the

PM_1 quadtree because in the PMR quadtree there is no need to subdivide in order to separate line segments that are very "close" or whose vertices are very "close," which is the case for the PM_1 quadtree. This is important since four blocks are created at each subdivision step. Thus when many subdivision steps occur, many empty blocks are created and thus the storage requirements of the PMR quadtree are considerably lower than those of the PM_1 quadtree. Generally, as the splitting threshold is increased, the storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase (see Section 6). Another advantage of the PMR quadtree over the PM_1 quadtree is that by virtue of being edge based, it can easily deal with nonplanar graphs.

It is interesting to point out that although a bucket can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown [Same90a] that the maximum number of line segments in a bucket is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block).

The PMR quadtree (and also the PM_1 quadtree) can be easily adapted to deal with fragments that result from set-theoretic operations such as union and intersection so that there is no data degradation when fragments of line segments are subsequently recombined. This is a direct consequence of spatial indexing — i.e., each block contains a descriptor of the object that is associated with it rather than the actual part of the object that occupies the block. The result is a consistent representation of line fragments since they are stored exactly and, thus, they can be deleted and reinserted without worrying about errors arising from the roundoffs introduced by approximating their intersection with the borders of the blocks through which they pass.

5 ALGORITHM TO FIND THE NEAREST LINE SEGMENT TO A POINT

Users of spatial databases frequently require the determination of the nearest object to a specified point or object. In computer graphics this is known as a "pick" operation where the query point or object corresponds to the location of a pointing device such as a cursor or a mouse. The utility of this operation becomes apparent when we observe that users of graphical interfaces often find it difficult to position a pointing device directly on top of an object such as a line segment for the purpose of selecting it.

We examine the problem of finding the nearest line segment to a point P. The "nearest" line segment is the one whose Euclidean distance to P is minimal. The collection of line segments is represented using a PMR quadtree. The first step in our algorithm is to locate the smallest block that contains P. We use the term *base region* to describe this block. Two items are worthy of note. First, the base region can be empty (e.g., the SW block in the SW quadrant in Figure 3(e)). Second, it should be clear that the nearest line segment to P need not be in the base region. For example, in Figure 3(e) line segment h is closer to query point P than line segments d and i which both pass through the

base region. Thus, in order to determine the nearest line segment to a query point, it is not sufficient to merely search the base region or its three brothers. In particular, when distance is measured using the Euclidean distance metric (as it is here), we find that in the worst-case, we must examine blocks that are not immediately adjacent to the base region.

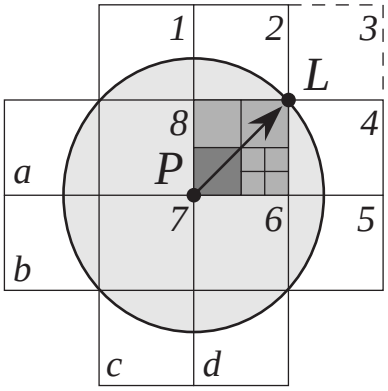

Figure 4: Blocks comprising the search region for finding the nearest line segment.

We use a four stage search process illustrated by Figure 4. Assume that the splitting threshold is k and that P is the query point. Let L be the location of a very small line segment, represented here as a point. The darkest gray area is the base region while the medium gray area includes the three brothers of the base region. In the worst case, P and L are found at opposite corners of the diagonally adjacent brothers. We know that there must exist at least k line segments in the union of the four brother blocks as otherwise (i.e., if there were less than k line segments) the four brothers would have been merged together to form a single block. As there must exist more than k line segments in the union of the base region and the three other brothers, we know that there also must exist a line segment whose distance to the specified point is less than or equal to the length of the diagonal across the four brothers[2].

Therefore, we see that the maximum value of the search radius is equal to the length of the diagonal across the block corresponding to the parent of the base region. By using this maximal distance as a radius, we form a circular search region centered at P. This constrains the search region to be cross-shaped where the interior is the subject of the first three stages, while the exterior is the focus of the fourth stage.

The first stage examines the block containing the query point (i.e., the base region) to determine the closest line if one exists here. If such a line exists, then the distance to it serves as the *initial search radius* (we use this term in our analysis of the experimental results in Section 6). In the second stage, we examine the brothers of the base region to see if they contain a closer line segment. If no line segment was found in the first stage, then the distance to the first line segment found in this stage serves as the *initial search*

[2]This constraint is not true when we are dealing with queries of the third class. In particular, if the splitting criteria are independent of the type of line segment, then the nearest line segment in the constraining search region may not necessarily be the one we are seeking. Such queries are a subject of future work.

radius. In the third stage we examine the blocks of size equal to (or greater than) the parent block, say T, of the base region that are immediately adjacent to T. There are at most eight such blocks and at most seven of them need to be examined (e.g., blocks 1, 2, and 4–8 in Figure 4). Stage four searches at most four additional blocks of size equal to (or greater than) T, if necessary (e.g., blocks a, b, c, and d in Figure 4).

Of course, not all four stages are executed in their entirety nor are all of the blocks examined. As line segments are found, the maximum search radius value is adjusted and used as a filter to avoid needless searches in many of the blocks. Given an area of s for the base region, in the worst case, we must search entirely or partially a total of twelve regions of size equal to that of the parent block where the total search area is equal to $8\pi s$ (i.e., $\pi(2\sqrt{s+s})^2$).

The execution time of the algorithm is proportional to the value of the splitting threshold and is independent of image resolution and complexity. There are several ways to obtain this result. One approach is to make use of the same techniques that were employed to analyze algorithms for operations on region quadtrees that made use of neighbor finding (e.g., [Same90b]). This requires an appropriate image model.

An alternative analysis is as follows. Assume that the line segments are uniformly distributed over the entire map region. In the absence of severe clustering, the degree of subdivision for two regions of the same size will be roughly equivalent. As can be seen from Figure 4, the maximal search radius will result in examining at most 32 blocks of size equal to the base region. If we assume that each block will contain $k/2$ line segments on the average, then we can reasonably expect that fewer than $32 \cdot k/2$ line segments will be considered as the closest possible line segment. Therefore, the average case computational complexity of the algorithm is $O(k)$.

6 EXPERIMENTAL RESULTS

Experiments were run using TIGER/Line file maps (see Figure 5 for an example map). All tests were executed on a Sun SparcStation 1+ (roughly 16 MIPs). The collections of line segments comprising the maps were stored in a PMR quadtree within the QUILT geographic information system developed at the University of Maryland [Shaf90]. In QUILT, the PMR quadtree is implemented using a pointerless quadtree. This representation is designed for large databases that are disk resident and this is the way we conducted our experiments. The implementation uses a linear quadtree where the blocks are sorted using bit interleaving or a z-order (e.g., [Trop81, Oren84]) and then stored in a B-tree. The maximum level of decomposition of each map is fourteen (i.e., the line segment database was normalized to lie in a $2^{14} \times 2^{14}$ region).

We were interested in answering the following questions:

- What is the relationship between map segment density and the search times for finding the nearest line segment?
- What is the average execution time for our implementation of the algorithm to find

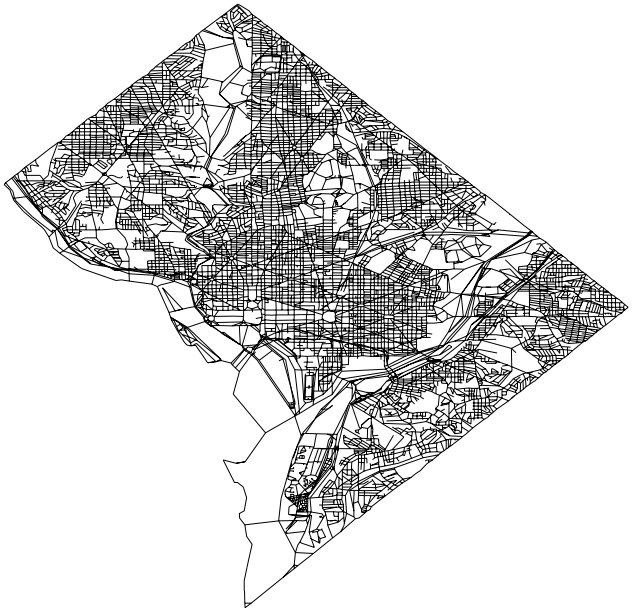

Figure 5: Washington D.C.

the nearest line segment?

- What is the effect of changing the value of the splitting threshold on the execution time of the algorithm to find the nearest line segment?
- What is the effect of changing the value of the splitting threshold on the storage requirements of the PMR quadtree?

In order to obtain this information, we gathered data on the average execution time per query, the average number of segments that were tested as candidates for the closest line segment, the initial search radius value (i.e., the distance between the query point and the nearest line segment found in its parent block), and the ratio of the initial search radius value to the maximal possible search radius (i.e., the length of the diagonal across the parent block). The effects of varying the splitting threshold on the storage requirements of the PMR quadtree and on the execution time of the nearest neighbor algorithm were also tabulated. Of secondary interest to these four questions, was a desire to obtain some knowledge about actual line segment data and how the PMR quadtree handles it. Data was collected on the total number of line segments, number of blocks, and the expected number of q-edges in each block.

Our initial tests were done using random test points that were uniformly distributed. This resulted in a high proportion of the query points being located 'outside' of the county boundaries. An extreme example was the San Francisco map where over 95% of the area represented by the PMR quadtree is void of line segments (essentially regions that do not lie within the county boundaries). Thus, a large majority of all uniformly distributed random query points were located outside of the county boundaries. This, of course, caused inflated estimates of the average execution times as the initial search radii were frequently very large.

In an attempt to more accurately correlate the query points with the data density (i.e., regions with high concentrations of line segments are more likely to be queried than sparse regions), we used a two stage process to generate our query points. We first generated the PMR quadtree block at random using a uniform distribution based on the total number of blocks—not their size. Next, having obtained a random block, we generated a query point at random within the block. In this case, we did draw the coordinates of the query point from a uniform distribution. For each of the listed maps, we ran 5,000 test queries.

Tables 1 and 2 contain a summary of our results by county and city. The data was generated from a PMR quadtree with a splitting threshold of eight. The q-edges in each block were examined sequentially—they were not sorted. The figures in parentheses are standard deviations. For most of the maps we have two entries because the spatial and nonspatial information in TIGER/Line maps is contained in two files for each governmental entity such as a county or a city. The first is called the *Basic Data Record* and it contains a single data record for each unique feature segment. A basic data record might represent a physically curved street. The shape of the street within the basic data record is approximated as a single line segment. The second file is called the *Shape Coordinate Points* and it contains additional coordinate points that lie between the two original endpoints of the basic data record. This allows the shape of the street to be more accurately represented, if necessary. We used both of these files in our tests. The first file is identified in the tables by appending the suffix α to the name of the map, while the second file is identified by the suffix β.

		quadtree statistics			*average per query*			
County		*seg count*	*blocks*	*q-edges per block*	*cpu seconds*	*segment comparisons*	*init srch radius*	*ratio of max search radius*
Arlington, VA	α	7237	3616	4.69	0.0149 (0.0045)	33.07 (7.90)	74.54	0.1586 (0.021)
	β	8333	4117	4.60	0.0147 (0.0036)	33.04 (8.40)	68.78	0.1628 (0.015)
Carroll, MD	α	8893	5011	4.66	0.0150 (0.0050)	29.67 (6.57)	66.39	0.1660 (0.023)
	β	13861	7084	4.45	0.0154 (0.0047)	32.09 (7.17)	59.01	0.1659 (0.023)
Howard, MD	α	10381	5434	4.43	0.0141 (0.0032)	30.62 (5.32)	48.64	0.1648 (0.016)
	β	16387	7975	4.43	0.0151 (0.0043)	32.19 (6.92)	41.42	0.1667 (0.014)
Harford, MD	α	12142	6403	4.63	0.0146 (0.0023)	31.56 (4.76)	52.54	0.1671 (0.017)
	β	19897	9745	4.44	0.0161 (0.0044)	32.43 (6.71)	45.98	0.1724 (0.020)
Marin, CA	α	18452	9931	4.53	0.0147 (0.0043)	32.36 (5.18)	27.29	0.1669 (0.013)

Table 1: Nearest line segment performance by county

From Tables 1 and 2 we see that the average number of line segment comparisons for each query varies between 29.67 and 37.22 for all of the line maps. In relative terms, for each map, our search for the nearest line segment only examines between 0.16% and 1.33% of the total number of line segments. As the number of line segment comparisons is relatively constant across all maps, we found that the larger the number of line segments in the map, the smaller the percentage of line segments in the map that need to be considered as possible closest line segments.

The average execution times range from 0.0141 and 0.0185 seconds across all of the maps. The execution time does not appear to be correlated with the initial search radius value. However, the initial search radius value is very dependent on the density of the

City		*quadtree statistics*			*average per query*			
		seg count	*blocks*	*q-edges per block*	*cpu seconds*	*segment comparisons*	*init srch radius*	*ratio of max search radius*
Charlottesville, VA	α	2412	1207	4.65	0.0145 (0.0029)	32.29 (6.68)	160.22	0.1603 (0.013)
	β	2942	1426	4.53	0.0143 (0.0037)	31.99 (6.94)	147.54	0.1595 (0.021)
Petersburg, VA	α	2920	1480	4.68	0.0148 (0.0034)	32.49 (5.80)	131.87	0.1665 (0.024)
	β	3597	1759	4.59	0.0149 (0.0029)	32.03 (8.31)	109.51	0.1625 (0.020)
Alexandria, VA	α	3769	1864	4.65	0.0143 (0.0033)	32.93 (7.57)	102.30	0.1613 (0.021)
	β	4829	2314	4.58	0.0148 (0.0035)	33.66 (7.39)	98.24	0.1621 (0.019)
Richmond, VA	α	13222	6499	4.74	0.0155 (0.0036)	34.98 (7.46)	56.28	0.1618 (0.024)
	β	15146	7306	4.68	0.0154 (0.0040)	34.80 (6.93)	51.26	0.1575 (0.017)
Washington, DC	α	15994	7918	4.83	0.0156 (0.0044)	34.37 (7.79)	54.22	0.1587 (0.022)
	β	18321	8857	4.77	0.0147 (0.0041)	34.01 (8.00)	49.98	0.1605 (0.025)
San Francisco, CA	α	16069	8182	4.85	0.0185 (0.0037)	37.22 (8.41)	11.47	0.1630 (0.012)
	β	18898	9622	4.79	0.0165 (0.0035)	36.52 (8.38)	12.63	0.1639 (0.093)

Table 2: Nearest line segment performance by city

line segments in the map. In particular, we observe that the San Francisco map has by far the smallest initial search radius value, while the Charlottesville map has the largest initial search radius value. This was not surprising once we examined the maps and saw that San Francisco had the largest segment density of all the maps under consideration, while Charlottesville had the smallest.

When considering the ratio of the initial search radius value to the maximum search radius value, we found that there was very little variation across the maps (i.e., it varies between 0.1575 and 0.1724). This means that the initial search radius value is very well correlated with the size of the base region. Therefore, San Francisco has a small initial search radius value because it has many small blocks. Thus we see a good, but somewhat surprising, result that a larger initial search radius value does not harm the performance of the search since the surrounding blocks that we will search are also usually larger. Therefore, the average search behavior will be the same for all maps, regardless of the initial search radius value.

Figure 6 shows the execution times of the five city maps as a function of the splitting threshold. Instead of using absolute quantities, we use ratios with respect to the storage requirements for a splitting threshold of eight (it is shown as a unit value). The performance ratios are on the vertical axis, while the splitting thresholds and the city map names are on the two horizontal axes. Observe that as the splitting threshold decreases, the time necessary to determine the nearest line segment also decreases. It should be clear that the relationship between the splitting thresholds and the execution times appears to be nearly linear. This provides empirical confirmation of our earlier analysis that the algorithm to find the nearest line segment in a PMR quadtree is $O(k)$, where k is the splitting threshold. This relationship is not surprising as, in essence, we are exchanging storage for execution speed. In particular, as the splitting threshold decreases, fewer line segments are found in each block, and therefore less time is spent sequentially applying an operation to each segment in the block under consideration.

Figure 7 shows the storage requirements of five city maps as a function of the splitting

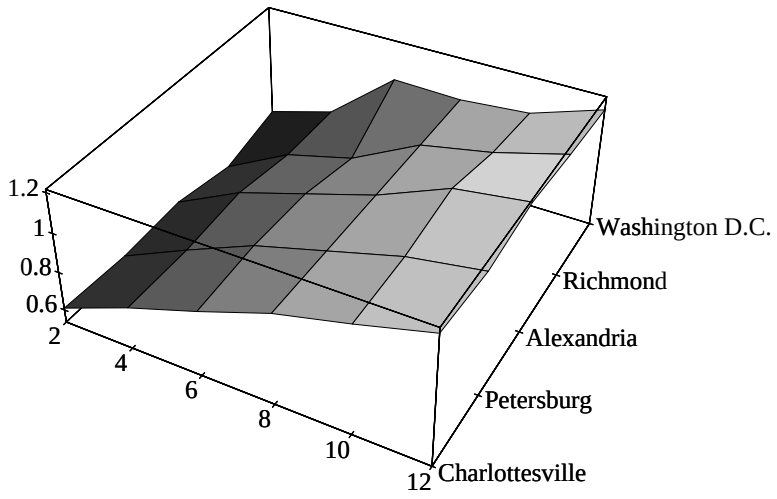

Figure 6: Nearest line segment execution time ratios for city maps by splitting threshold.

threshold. Again, instead of using absolute quantities, we use ratios with respect to the storage requirements for a splitting threshold of eight (it is shown as a unit value). The performance ratios are on the vertical axis, while the splitting thresholds and the city map names are on the two horizontal axes. We see clearly that as the splitting threshold decreases, the amount of storage necessary increases dramatically for each of the five maps. A splitting threshold of two requires at least twice as much storage for each map as was needed with a splitting threshold of eight. We also note that the proportional rate of increase in storage is slightly larger for the larger maps although it is difficult to see in the figure. In essence, we are observing that as the splitting threshold decreases, the number of blocks in the PMR quadtree is increasing because the capacity of the block is decreasing. In addition, line segments that span several blocks are inserted into more of the smaller blocks than when the threshold values were larger thereby obviating the need to split the blocks, which means that the blocks are larger.

The above results confirm our analysis that the execution time of the algorithm to find the nearest line segment is proportional to the splitting threshold. This is plainly evident from Figure 6. Tables 1 and 2 also support this analysis since neither the execution time nor the number of segment comparisons vary greatly across the different maps. The only outlier was San Francisco. This was easily explained once we saw its map since it has a tremendous amount of empty area and is not contiguous.

In our analysis we also assumed that each block would contain $O(k/2)$ line segments. This assumption is supported by Tables 1 and 2 where for a splitting threshold of eight, the average block in our test maps contained between 4.43 and 4.85 line segments. In fact, we also found that only rarely did the occupancy of a block exceed the value of the splitting threshold. In particular, for all the maps that we tested, with a splitting threshold of eight, less than 0.5% of the blocks contained more q-edges than the splitting threshold and the maximum that we observed was eleven. Recall that the theoretical maximum

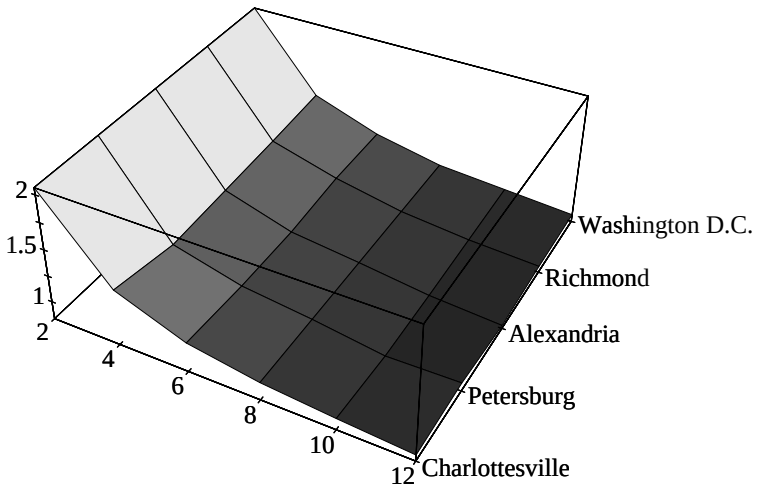

Figure 7: Storage requirement ratios for city maps by splitting threshold.

number of line segments in a block is equal to the sum of the splitting threshold and the depth of the block (i.e., 8+14=21 in our case).

7 CONCLUDING REMARKS

We have seen that efficient processing of spatial queries is an important problem in the implementation of spatial databases. Although our objects consisted only of line segments and the problem of finding the nearest line segment, the issue has wide applicability. Future work should examine additional queries. Unfortunately, it is difficult to identify them. This will not be discussed further here.

Our representation makes use of a bucketing approach which sorts the line segments with respect to the space from which they are drawn. We used an adaptive approach in contrast to a uniform grid where all the buckets are of the same size. We feel that this yields superior performance especially in empty regions since they are aggregated and hence the number of such regions that are visited is considerably lower than is the case with a uniform grid. Data such as the San Francisco map reinforces this view.

An obvious issue involves a more detailed examination of the effects of the bucket capacity (or more precisely the splitting threshold) on the storage and execution time requirements of the algorithms. Is there an ideal splitting threshold? Is there some obvious relationship between the resolution of the data (i.e., the maximum number of allowable subdivision steps), splitting threshold, and the volume of the data? Some of our other studies have shown that as the value of the splitting threshold is increased by several orders of magnitude (e.g., above 200), the execution time rises dramatically. This is a direct result of the fact that we do not sort the line segments within each bucket.

When the splitting threshold value is relatively small (e.g., under twelve), this is not a serious issue since sequential search is efficient. However, as the splitting threshold value is increased, the fact that each access to a bucket requires that it be searched sequentially is inefficient. We need to factor out the effect of sorting, or lack of it, in measurements that involve larger splitting threshold values. Of course, the issue of how to sort the line segments within the block is a problem in its own right.

Another issue involves the measurement of the performance of spatial algorithms. We need to investigate models for spatial data and then analyze the storage and execution time costs for them. The difficulty lies in making the models realistic so that the spatial objects that are generated by them have a distribution similar to that found in real data. For example, our experience has been that random line segments cannot be generated by simply choosing the coordinate of each endpoint from a uniform distribution [Jaga90]. Such an approach ignores the connectivity that is so often found in polygonal maps. An interesting area for further investigation is the possible use of geometric probability [Sant76] to generate the test data. However, once again, we must not ignore connectivity.

Similarly, we must develop appropriate models for generating the data necessary to measure the performance of the queries. This is different from the test spatial data discussed above. For example, when computing the nearest line segment to a point, we saw the futility of choosing the query points at random from a uniform distribution. In particular, such an approach will be biased towards generating query points in large empty regions. Our point is that the distribution of the actual spatial data should be used to test it—not some illusory distribution. This is the approach we followed in testing the computation of the nearest line segment to a given point. Recall that query points were generated using a two stage process. We first generated the block at random from a uniform distribution based on the number of blocks—not their size. Next, having obtained a random block, we generated a query point at random within the block. In this case we did draw the coordinates of the query point from a uniform distribution.

The generation of the block address at random is important. If we don't do this, then tests would reveal that a uniform grid data structure would be superior to a PMR quadtree which is fine if the actual data were uniformly distributed. However, as we saw from our test data, this is usually not the case (e.g., the San Francisco map).

The algorithms employed to respond to the queries also suggest avenues for future research. All of our proximity queries have measured distance in terms of the Euclidean metric. The Euclidean metric, although commonly used, has two drawbacks. First, its computation is time-consuming since a square root operation must be calculated. Second, by virtue of its locus being a circle, for a given distance value, the Euclidean metric causes more neighboring buckets to be examined than would a metric that is more attuned to the rectangular shape of the buckets (e.g., the Chessboard metric which is also known as the maximum value metric and whose locus is a square). Two interesting questions arise. First, for dense data, how often is the "approximate" closest line segment (obtained by using the Chessboard metric) different from the "true" closest line segment (obtained by using the Euclidean metric)? Second, what is the extra cost in using the Euclidean metric over the Chessboard metric?

References

[Bure89] Bureau of the Census, TIGER/Line precensus files: 1990 technical documentation, Bureau of the Census, Washington, DC, 1989.

[Buch90] A. Buchmann, O. Günther, T.R. Smith, and Y.-F. Wang, eds., *Design and Implementation of Large Spatial Databases*, Lecture Notes in Computer Science No. 409, Springer-Verlag, Berlin, 1990,

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

[Falo87] C. Faloutsos, T. Sellis, and N. Roussopoulos, Analysis of object oriented spatial access methods, *Proceedings of the SIGMOD Conference*, San Francisco, May 1987, 426–439.

[Fole90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley, Reading, MA, 1990.

[Fran84] W. R. Franklin, Adaptive grids for geometric operations, *Cartographica 21*, 2&3(Summer & Autumn 1984), 160–167.

[Free87] M. Freeston, The BANG file: a new kind of grid file, *Proceedings of the SIGMOD Conference*, San Francisco, May 1987, 260–269.

[Günt87] O. Günther, Efficient structures for geometric data management, Ph.D. dissertation, UCB/ERL M87/77, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, Berkeley, CA, 1987 (Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, 1988).

[Gutt84] A. Guttman, R-trees: a dynamic index structure for spatial searching, *Proceedings of the SIGMOD Conference*, Boston, June 1984, 47–57.

[Henr89] A. Henrich, H. W. Six, and P. Widmayer, The LSD tree: spatial access to multidimensional point and non-point data, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, P. M. G. Apers and G. Wiederhold, eds., Amsterdam, August 1989, 45–53.

[Hinr83] K. Hinrichs and J. Nievergelt, The grid file: a data structure designed to support proximity queries on spatial objects, *Proceedings of the WG'83 (International Workshop on Graphtheoretic Concepts in Computer Science)*, M. Nagl and J. Perl, eds., Trauner Verlag, Linz, Austria, 1983, 100–113.

[Jaga90] H. V. Jagadish, On indexing line segments, *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, D. McLeod, R. Sacks-Davis, and H. Schek, eds., Brisbane, Australia, August 1990, 614–625.

[Nels86] R. C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4(August 1986), 197–206 (also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

[Nels87] R. C. Nelson and H. Samet, A population analysis for hierarchical data structures, *Proceedings of the SIGMOD Conference*, San Francisco, May 1987, 270–277.

[Niev84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, *ACM Transactions on Database Systems 9*, 1(March 1984), 38–71.

[Oren89] J. A. Orenstein, Redundancy in spatial databases, *Proceedings of the SIGMOD Conference*, Portland, OR, June 1989, 294–305.

[Oren84] J. A. Orenstein and T. H. Merrett, A class of data structures for associative searching, *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, Canada, April 1984, 181–190.

[Peuq90] D. J. Peuquet and D. F. Marble, ARC/INFO: An example of a contemporary geographic information system, in *Introductory Readings In Geographic Information Systems*, D. F. Peuquet and D. F. Marble, eds., Taylor & Francis, London, 1990, 90–99.

[Same90a] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[Same90b] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.

[Same85] H. Samet and R. E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182–222.

[Sant76] L. A. Santalo, Integral geometry and geometric probability, in *Encyclopedia of Mathematics and its Applications*, G. C. Rota, ed., Addison-Wesley, Reading, MA, 1976.

[Shaf90] C. A. Shaffer, H. Samet, and R. C. Nelson, QUILT: a geographic information system based on quadtrees, *International Journal of Geographical Information Systems 4*, 2(April–June 1990), 103–131.

[Seeg90] B. Seeger and H. P. Kriegel, The buddy-tree: an efficient and robust access method for spatial data base systems, *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, D. McLeod, R. Sacks-Davis, and H. Schek, eds., Brisbane, Australia, August 1990, 590–601.

[Ston86] M. Stonebraker, T. Sellis, and E. Hanson, An analysis of rule indexing implementations in data base systems, *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, April 1986, 353–364.

[Tamm81] M. Tamminen, The EXCELL method for efficient geometric access to data, *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34, Helsinki, Finland, 1981.

[Tamm82] M. Tamminen, Efficient spatial access to a data base, *Proceedings of the SIGMOD Conference*, Orlando, June 1982, 47–57.

[Trop81] H. Tropf and H. Herzog, Multidimensional range search in dynamically balanced trees, *Angewandte Informatik 23*, 2(February 1981), 71–77.