In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178-189, Charleston, SC, August 1992.

UNIQUELY REPORTING SPATIAL OBJECTS: YET ANOTHER OPERATION FOR COMPARING SPATIAL DATA STRUCTURES[1]

Walid G. Aref
Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
The University of Maryland, College Park, Maryland 20742
E-mail: aref@umiacs.umd.edu, hjs@umiacs.umd.edu

Abstract

Many spatial data structures (e.g., the quadtree, the cell tree, the R^+-tree) represent an object by partitioning it into more than one piece, each of which is stored separately inside the data structure. Frequently, it is required to report the objects stored in a particular subset (spatially defined) of the database. It can be achieved by a simple traversal of the spatial data structure. This may result in reporting each object as many times as the number of partitions of this object inside the data structure. An operation, termed Report_Unique, is defined which reports each object in the data structure just once, irrespective of the multiplicity of the partitions of the object. Example algorithms are presented to perform Report_Unique for a quadtree. A classification of spatial objects is also presented and it is shown to affect the complexity of performing the Report_Unique operation.

1 Introduction

There are several ways of representing a spatial object inside a data structure [6]. Some data structures (e.g., the R-tree [4], the Grid file [8]) represent a spatial object by just one entity inside the data structure (e.g., by the object's bounding rectangle in the case of the R-tree). On the other hand, another group of data structures (e.g., the quadtree [5], the cell tree [3], the R^+-tree [2]) represent a spatial object by partitioning it into more than one piece, each of which is stored separately inside the data structure. In this paper, we focus on the latter group of data structures.

One of the most important functions of a spatial database system is to report the spatial objects in a given portion of the database (e.g., the objects inside a query window). One way to do this is to traverse the underlying data structure and to report each object encountered. However, this simple traversal of the spatial data structure may result in reporting each object as many times as the number of partitions of this object inside the data structure. When a spatial object is represented by more than one piece inside the data structure, this poses a problem since the object will be reported more than once. In this paper we define an operation, termed Report_Unique, which reports each object in the data structure just once, regardless of the multiplicity of the partitions of the object inside the spatial data structure.

The Report_Unique operation has many applications. For example, suppose we want to report the spatial objects that lie inside a window, say w. Although several partitions of the same object,

[1]The support of the National Science Foundation under Grant IRI-9017393 is gratefully acknowledged.

say o, may lie inside w, it is preferable to report the identity of o just once. This is especially true if the identifier of o is to participate in further processing, e.g., if the identifier of o serves as the argument to another procedure, say p (e.g., for computing the area or perimeter of the object). In this case, if the identifier of o is reported more than once, then o will be processed by p redundantly. Given the data structure holding the spatial objects that lie inside the window, Report_Unique ignores the multiplicity of the object partitions inside the data structure and passes each object in the data structure to p just once.

As an example of another application of Report_Unique, suppose we want to count the number of spatial objects in the given data structure. In this case, we need to count each object only once regardless of the number of partitions of the object.

The rest of the paper proceeds as follows. In Section 2 we outline a straightforward algorithm to solve the Report_Unique problem and analyze its worst-case space and time complexities. The straightforward approach requires $O(n)$ space where n is the number of spatial objects in the database. This may be unacceptable for large databases. Section 3 describes another algorithm that eliminates the space requirements of the algorithm of Section 2. Section 4 points out that supporting the Report_Unique operation for a realistic variety of objects requires classifying spatial objects into categories of different complexity, and uses line segment objects as an example. Section 5 shows how these classes affect the performance and correctness of the alternative approaches to Report_Unique for line segment objects. Section 6 contains concluding remarks.

It is important to note that in this paper we do not attempt to compare the efficiency of different data structures in performing Report_Unique nor do we present efficient algorithms for each case. Instead, our emphasis is on demonstrating the importance of Report_Unique when considering new data structures, and the need for efficient algorithms, especially when dealing with large spatial databases.

2 The Test-And-Set Algorithm

A straightforward algorithm for Report_Unique is the test-and-set algorithm (or Report_Unique1). It works for almost all data structures that partition a spatial object into smaller pieces, provided that each spatial object has a unique identification number that is stored in the data structure along with each piece of the object. The algorithm makes use of an array of bits. There is one bit for each object which is initially false. The algorithm proceeds as follows:

1. Traverse the data structure, and for each sub-object encountered test the object's corresponding bit in the array.
2. If the bit is false, then set it to true and retrieve the object using the object's identifier and report it.
3. If the bit is true, then don't report this object.

Assume that there are n spatial objects stored in the data structure and that on the average each object is partitioned into k pieces inside the data structure.[2] Assume further that the number of data structure elements visited during the traversal is m (e.g., m quadtree nodes). Then, the test-and-set algorithm takes n bits for storing the bit array, performs kn tests and n object retrievals (possibly from secondary storage). Therefore, the total cost of the algorithm is $C_m + knC_{cpu} + nC_{io}$, where C_m is the cost of traversing the m elements of the data structure, C_{cpu} is the cost of testing a bit, and C_{io} is the cost of retrieving the spatial object (e.g., the coordinate values of a line segment, the polygon boundary, etc.). Since all the algorithms that we present are based on traversal, we will omit the term C_m from the comparison. Therefore, the cost of the algorithm is $knC_{cpu} + nC_{io}$. This component of the cost cannot be ignored for algorithms that are not based on traversal of the data structure (e.g., direct access of data structure elements).

[2]Notice that the value of k varies from one data structure to another.

In order to improve the execution time or space requirements of Report_Unique1 we observe that the Report_Unique operation requires a test function, say t, such that given a spatial object, say o, with k pieces, p_1, p_2, ..., p_k, t has a value of false for only one of the pieces, say p_j of o. Application of t to the rest of the pieces yield the value of true. Any function t that satisfies this criterion can be used as a way of suppressing all pieces of o other than p_j from reporting o's object identifier, and hence can be adopted by the Report_Unique operation.

For example, in the case of the test-and-set algorithm, t_1 is defined as follows (a is the bit array and $oid(p_i)$ is the identifier of the object that piece p_i belongs to):

$$\begin{aligned} t_1(p_i) \quad = \quad & ret_value \leftarrow a[oid(p_i)]); \\ & if(not\ a[oid(p_i)])\ then \\ & \quad a[oid(p_i)] \leftarrow true; \\ & return\ ret_value; \end{aligned}$$

3 Avoiding the Extra Space

As an example data structure, consider the PMR quadtree [7]. In this data structure, a line segment is divided into several pieces, where each piece is associated with the quadtree block that it intersects. Figure 1 shows one example of the PMR quadtree. Each block stores the object identifiers of the

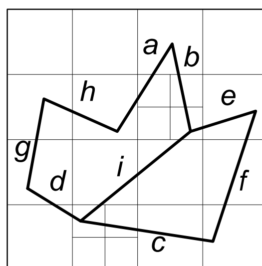

Figure 1: An example of a PMR quadtree.

lines passing through it. On the other hand, the full description of the line segments (e.g., the start and end coordinate values of the points) is stored in what is called the *feature table*. The index of line l in the feature table is l's object identifier. Notice that we cannot simply traverse the feature table and report the lines that we find since not all the lines in the feature table have to belong to the data structure in question.

Our first attempt as improving on the test-and-set algorithm is to get rid of the $O(n)$ space requirements of the algorithm. This gives rise to algorithm Report_Unique2 described below. It traverses the blocks of the PMR quadtree, and for each block, say B, it follows the following actions:

1. For each object identifer i stored in B (that corresponds to piece p_i), retrieve the line description, say l_i, from the feature table, and
2. perform the following testing function t_2

$$\begin{aligned} t_2(l_i, B) \quad = \quad & if(start_point(l_i)\ in\ B)\ then\ return\ (true) \\ & else\ return\ (false) \end{aligned}$$

3. If $t_2(l_i, B)$ is true, report line l_i.

The algorithm is illustrated in Figure 2. Notice that for each line, say l, the testing function t_2 will

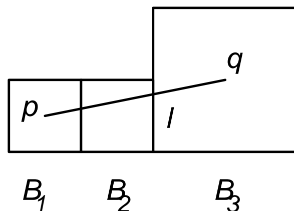

Figure 2: Line l is stored in blocks B_1, B_2, and B_3 where only block B_1 causes the point-in-block test to succeed since the starting point p of l lies inside B_1 only but not in B_2 or in B_3.

be true only when the block containing the starting point of l is processed. All blocks containing pieces of l other than the starting point of l will evaluate to false as the result of applying t_2 to them (t_2 is referred to as the point-in-block test).

A major advantage of this algorithm over the test-and-set algorithm is that it does not require any additional space. However, algorithm Report_Unique2 needs more time to execute. Note that it applies a more complex test per object piece. The testing function is still performed nk times. However, the point-in-block test requires four comparisons (each of cost C_{cpu}) in contrast to just one comparison in the test-and-set algorithm. Notice that the point (x, y) is inside block $B((bl_x, bl_y), (ur_x, ur_y))$, where bl and ur denote the bottom left and upper right corners of the window, respectively, *iff* $bl_x \leq x \leq ur_x \ \wedge \ bl_y \leq y \leq ur_y$.

A more serious drawback of the algorithm is that in order to perform the point-in-block test (t_2) we must access the feature table (via the object identifier) each time we encounter a piece of a line. This is necessary to retrieve the starting point of the line. Assuming that the cost of retrieving a segment from the feature table is also C_{io}, then the execution time of this algorithm is $4knC_{cpu} + knC_{io}$.

The cost term knC_{io} involves redundant disk-I/O requests. Basically, it represents the cost of retrieving line segments from the feature table (once for every line segment partition, amounting to k disk requests per line segment). A better algorithm would perform only n such requests, i.e., one request per line segment in the database instead of one request per line segement partition. Thus, we would like an algorithm that does not need the $O(n)$ extra storage (or at least having an asymptotic storage cost less then $O(n)$), yet still performs only n accesses to the feature table.

In Section 5, a third algorithm (Report_Unique3) is presented that overcomes this drawback. It uses the concept of an *active border* [9] to avoid accessing the feature table each time a piece of the line is encountered.

4 Object Classification

An important factor affecting the performance of the Report_Unique operation is the shape of the objects stored in the underlying data structure. For example, what is the effect of restricting the lines to be rectilinear, in contrast to lines with arbitrary slopes? As another example, suppose that objects are partially clipped as is the situation after a window operation. Is it more difficult to report these clipped objects than reporting non-clipped objects?

As an illustration of the second example, note that due to the way the PMR quadtree is defined, Report_Unique2 does not work properly if the line segments are partly clipped. This is shown in in Figure 3. In particular, when a line is clipped (e.g., as a result of a window operation), the PMR quadtree does not update the starting and ending points of the clipped line in the feature table. As a result, if a line, say l, is clipped so that only s remains and if s does not contain l's starting point, then s will not be reported by Report_Unique2. This case does not arise in Report_Unique1, and suggests that we have to consider the alternative classes of objects (e.g., clipped objects) when developing algorithms for Report_Unique since it affects the correctness of the algorithm.

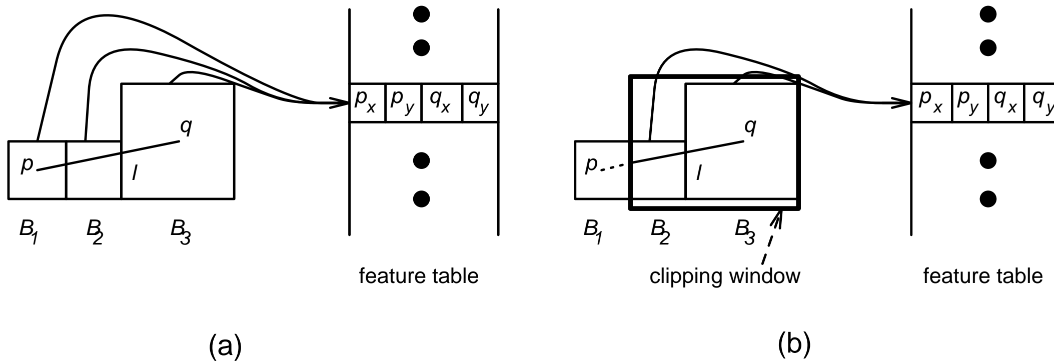

Figure 3: (a) Line l as stored in a PMR quadtree, (b) The result of clipping line l by a window operation in a PMR quadtree. Line l is partially clipped so that its original starting point in the feature table is neither in blocks B_2 nor B_3. Notice that the feature table still stores point p as part of the clipped line description.

Below, we present one way to classify spatial objects of type line segment. Other spatial types can be classified in an analogous manner. The different classes of line segments are given in Figure 4. The classes do not always impose different complexity requirements on Report_Unique. Example

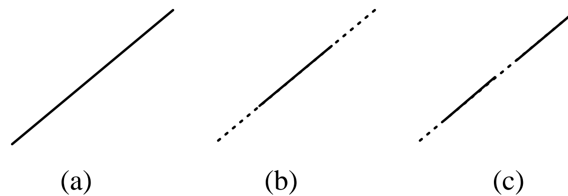

Figure 4: Classification of line segments (a) Class-1: a regular line segment, (b) Class-2: a clipped line segment, (c) Class-3: a broken line segment.

implementations are given in Section 5.

- Class-1 segments: a line segment having no clipped or missing portions as illustrated in Figure 4a, and termed a *regular* line segment.
- Class-2 segments: a line segment where either one or both of its end-points are missing as illustrated in Figure 4b, and termed a *clipped* line segment.
- Class-3 segments: a line segment where several portions (holes) may be missing as illustrated in Figure 4c, and termed a *broken* line segment. Although disjoint, all the portions of the line segment refer to, and represent, just one object of type line segment.

This classification of line segments is of practical use. Class-1 represents regular line segments (e.g., road segments). Class-2 represents line segments that may result from a rectangular window operation (Figure 5a). Class-3 represents line segments that may result from intersecting line segments with arbitrary regions (Figure 5b). Similar classes can be constructed for other spatial objects (e.g., polygons).

We further classify lines of each class into the following types according to their orientation:

- Type-1 segments: uniformly oriented line segments - i.e., they are parallel to the x or the y axes but not both (Figure 6a).

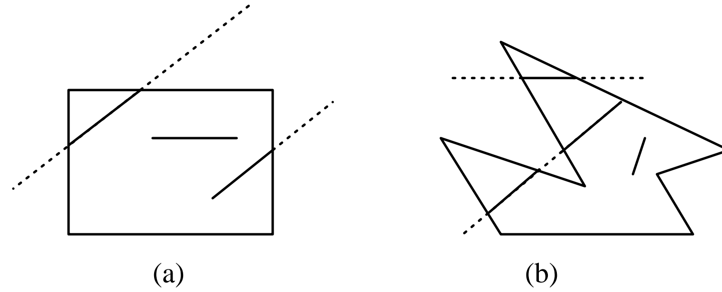

Figure 5: (a) Class-2 line segments result from clipping line segments via a window operation, (b) Class-3 line segments result from intersecting regular line segments with a simple polygon.

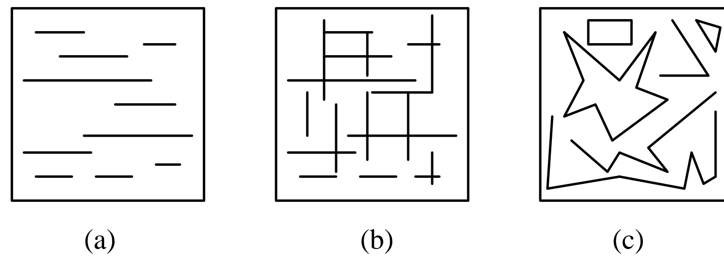

Figure 6: Three types of line segments: (a) line segments parallel to the x-axis, (b) rectilinear line segments, (c) line segments with arbitrary orientation.

- Type-2 segments: rectilinear line segments - i.e., they may be parallel to either the x or the y axes (Figure 6b).
- Type-3 segments: arbitrary line segments - i.e., they have arbitrary orientations (slopes) (Figure 6c).

5 Algorithms

Below, we present some algorithms that correspond to several class/type combinations given in Section 4 and see how the complexity of the algorithms are affected. Our underlying spatial database makes use of a PMR quadtree to store the spatial database objects which are line segments in these examples.

5.1 Class-1 Type-1 Line Segments

Assume a collection of line segments that are parallel to the x-axis (Figure 6a). They may represent time intervals for a group of events. The algorithm traverses the quadtree block by block and maintains the notion of the *active set* of line segments. A line segment, say l, is added to the active set when l is first encountered during the traversal. l is deleted from the active set once all the quadtree blocks that overlap with l have been visited during the traversal. The storage necessary is bounded by the maximun size of the active set during the execution of the algorithm. The size of the active set should be considerably smaller than n, the number of line segments in the spatial database.

In order to execute the algorithm efficiently, we need to organize the active set of line segments. The algorithm performs the following three primitive operations on an active set, say A: (1) test if

a line segment exists in A (notice that this is needed in order to decide whether a line has already been encountered or if it is its first encounter in the traversal), (2) insert a line segment into A, and (3) delete a line segment from A.

Before providing more details about the algorithm, observe that if we are able to perform each of the above three operations in $O(1)$ time and maintain the underlying data structure that stores the active set in $O(1)$ time, then the overall complexity of the algorithm would be $O(nk)$ time with $O(\textit{active set size})$ space. The size of the active set in the worst case is $O(bT)$ where b is the maximum number of spatial objects that can be stored in a quadtree block before it overflows (i.e., the $b$ucket size) and T is the width of the underlying spatial space. In practice, it is expected that the size of the active set is considerably smaller than $O(bT)$.

The algorithm visits the blocks of the quadtree in NW, NE, SW, SE scanning order. It maintains one basic data structure: an *active border* [9]. The active border represents the border between those quadtree blocks that have been processed and those that have not. The elements of the active border form a "staircase" of vertical and borizontal edges, moving from southwest to northeast, as shown by the heavy line in Figure 7a. Initially, the active border consists of the north and west borders of

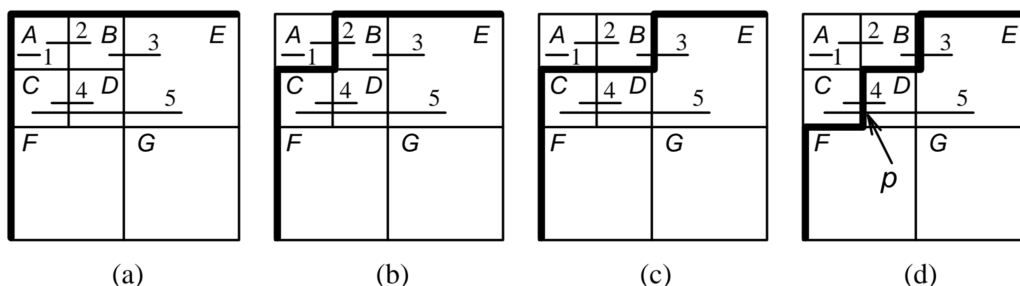

Figure 7: Various states of the active border: (a) initial state, (b) after visiting block A, lines 1 and 2 are reported, line 1 is deleted, (c) after visiting block B, line 3 is reported, line 2 is deleted, (d) after visiting block C, lines 4 and 5 are reported.

the image. When the algorithm terminates, the active border consists of the south and east borders of the entire image. In other words, whenever a node is visited by the algorithm, the active border is updated accordingly to include the border of this new block (see Figures 7b and 7c). Each element of the active border stores in it the set of active line segments that intersect the portion of the active border corresponding to this element. For example, in Figure 7d, lines 4 and 5 are associated with portion (element) p of the active border.

When a leaf block is processed, the portion of the active border that is adjacent to the block must be located. In [1], a technique is developed where blocks can be located in the active border in constant time. This is achieved by traversing (and updating) the active border along with the quadtree traversal while passing and stacking pointers to guarantee that the exact location in the active border is available ($O(1)$ time) whenever needed, so that searching in the active border is entirely avoided. The reader is referred to [1] for further details. Once the appropriate active border element is located, testing for existence, insertion, and deletion of a line segment into this element takes $O(b)$ time which is really a constant or $O(1)$.[3]

We define an active line segment as a line segment that is partially processsed, i.e., at least one of the quadtree blocks overlapping with this line has not been processed yet. A line is inactive if either all of its overlapping blocks have been processed by the traversal algorithm or if all of them are yet to be processed. Notice that a line segment is reported once it is first inserted in the active border, and is deleted once its identifier does not appear in the neighboring quadtree of an active

[3]In some applications where the bucket size b is large, performing a linear search in $O(b)$ time tends to be too slow. In such a case, the objects inside each bucket can be sorted among themselves to achieve better search time (e.g., $O(\log b)$) within the bucket.

border portion that contains the line's identifier. For example, when block B is processed in the quadtree of Figure 7c, line 2 is deleted since it is entirely covered by the processed blocks A and B.

5.2 Line Segments of Classes 2 and 3

The algorithm of Section 5.1 can be extended easily to support Class-2 and Class-3 objects. In fact, for Class-2 Type-2 objects, the same algorithm applies correctly without any changes. For Class-3, a portion of a line segment can be hidden so that its absence from a neighboring block to the active border does not imply that this line segment has been processed in its entirety, and hence it cannot be deleted from the active border (see Figure 8). For this reason, the deletion mechanism of a line in

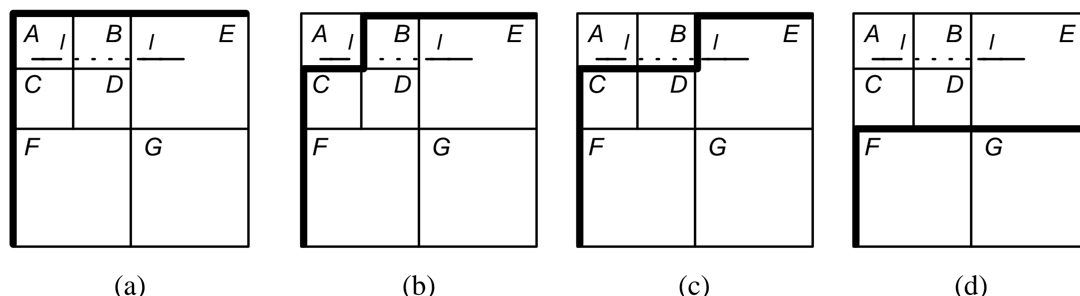

Figure 8: Example illustrating the difficulty with Class-3 Type-2 line segments. (a) initial state, (b) l is reported when block A is processed, (c) l is deleted when block B is processed, (d) l is reported again when block E is processed.

the algorithm of Section 5.1 does not work properly. In fact, if used, it may result in a line segment being reported several times since it was deleted erroneously.

In order to avoid this problem, additional information must be stored in the active border for each active line segment. When the identifier of a line segment, say l, is encountered for the first time during the traversal, l's coordinates are retrieved from the feature table. The end-point, say p, of l that is not covered yet by the traversal is stored in the active border along with l's identifier. Now, if a neighboring quadtree block, say B, is visited, where B is a neighbor of the portion of the active border that contains l's identifier, then we need to check whether p lies inside B or not. If yes, then l is deleted from the active border (l is entirely processed by now). If not, then l's identifier is propagated into the portions of the active border that represent block B. This process is illustrated in Figure 9.

5.3 Class-1 Type-3 Line Segments

The algorithm of Section 5.2 does not work for Type-3 objects. For example, consider Figure 10. This algorithm would report line segment l more than once, i.e., once for each of the visits of blocks C, D, and F. In addition, line segment s is also reported more than once - i.e., once for each of the visits of blocks E and F. The problem is worse for line segments of Classes 2 and 3.

This problem can be overcome by using a variant of the algorithm described in [1]. That algorithm computes the boundaries of regions in a region quadtree. We can adapt this algorithm to work for line segment objects by considering the regions through which the line segments pass. In particular, the variant reports the coordinate values of the line segment end-points passing through the region instead of the boundary. This version executes in $O(kn \cdot \alpha(kn))$ where $\alpha()$ is the inverse of Ackerman's function (a function that grows very fast and hence its inverse grows very slowly).

Note that the algorithm of [1] maintains the partial active border of a spatial object. This information may not be needed in the case of Report_Unique. This suggests that a simpler algorithm

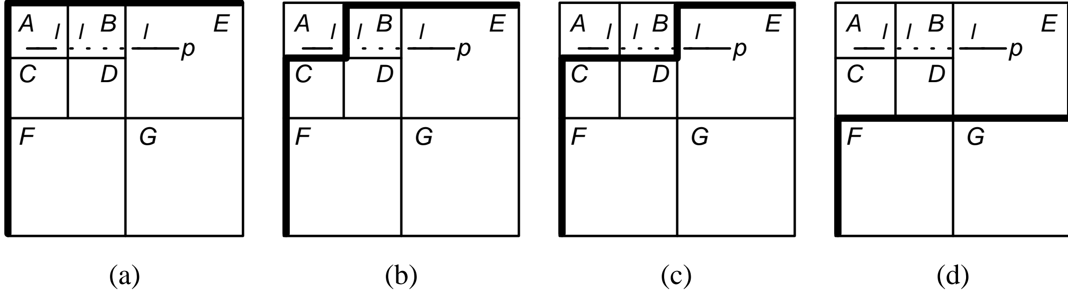

Figure 9: Class-3 Type-2 line segments. (a) initial state, (b) l is reported when block A is processed, (c) l is still considered active when block B is processed (although it does not overlap with B), (d) l is not reported when block E is processed, and is deleted from E afterwards since l's end-point p lies inside E.

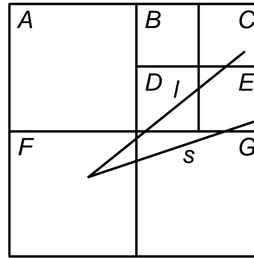

Figure 10: Example illustrating the difficulty with Class-1 Type-3 line segments.

(from a computational complexity standpoint) may be possible. We do not address this issue in this paper.

6 Conclusion and Future Research

Table 1 represents a summary of our results. It shows the different complexities of the algorithms developed for the class/type object combinations explored here. As we can see, Report_Unique is a challenging problem. Future work involves: developing better algorithms for other class/type combinations, classifying spatial objects besides line segments in an analogous matter and developing algorithms for them as well, and considering spatial data structures that partition objects in addition to the PMR quadtree.

References

[1] M. B. Dillencourt and H. Samet. Using topological sweep to extract the boundaries of regions in maps represented by region quadtrees. *Submitted for publication*, 1991.

[2] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 426–439, San Francisco, May 1987.

[3] O. Günther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 598–605, Los Angeles, February 1989.

	Type-1	Type-2	Type-3
Class-1	$O(kn)$ cpu, $O(n)$ i/o $O(T)$ space	$O(kn)$ cpu, $O(n)$ i/o $O(T)$ space	$O(kn)$ cpu, $O(n)$ i/o $O(n)$ space
Class-2	$O(kn)$ cpu, $O(n)$ i/o $O(T)$ space	$O(kn)$ cpu, $O(n)$ i/o $O(T)$ space	$O(kn)$ cpu, $O(n)$ i/o $O(n)$ space
Class-3	$O(kn)$ cpu, $O(n)$ i/o $O(T)$ space	$O(kn)$ cpu, $O(n)$ i/o $O(T)$ space	$O(kn)$ cpu, $O(n)$ i/o $O(n)$ space

Table 1: Summary of time and space worst-case complexity of Report_Unique for class/type combinations of line segment objects.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.

[5] A Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.

[6] H. P. Kriegel, P. Heep, S. Heep, M. Schiwietz, and R. Schneider. An access method based query processor for spatial database systems. In G. Gambosi, M. Scholl, and H.-W. Six, editors, *Geographic Database Management Systems. Workshop Proceedings, Capri, Italy, May 1991*, pages 194–211, Berlin, 1992. Springer-Verlag.

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

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

[9] H. Samet, A. Rosenfeld, C. Shaffer, R. Nelson, Y. Huang, and K. Fujimura. Application of hierarchical data structures to geographical information systems: Phase IV. Technical Report CS-1578, Univ. of Maryland, College Park, MD, December 1985.