CAR-TR-671
CS-TR-3067

DAAB10-91-C-0175
IRI-9017393
May 1993

DUPLICATE ELIMINATION USING PROXIMITY IN SPATIAL DATABASES

Walid G. Aref*
Hanan Samet*,†

*Center for Automation Research
*Computer Science Department
†Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742-3275

Abstract

A spatial database allows spatial objects to be indexed. In a spatial database, an object may extend arbitrarily in space. As a result, many spatial data structures (e.g., the quadtree, the cell tree, the R^+-tree) represent an object by partitioning it into multiple, simple pieces, each of which is stored separately inside the data structure. Many operations on these data structures are likely to produce duplicate results because of the multiplicity of object pieces. A novel approach for eliminating duplicate results based on proximity of spatial objects is presented. 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 indexed by the underlying representation. Example algorithms are presented that perform Report_Unique for a quadtree representation of line segments and for the more general case of arbitrary rectangles. The complexity of the Report_Unique operation is seen to depend on a geometric classification of different instances of the spatial objects. Such classifications are presented for spatial objects consisting of line segments, as well as rectangles.

Keywords: spatial databases, design and analysis of algorithms, duplicate elimination, data structures, quadtrees

The support of the Army at Vint Hill under Contract DAAB10-91-C-0175, as well as the support of the National Science Foundation under Grant IRI-9017393 is gratefully acknowledged, as is the help of Sandy German in preparing this paper.

1 Introduction

Spatial databases are large in data volume and complex in the inter-relationships between the constituent data. In this respect, a flexible and efficient spatial data organization can make a significant contribution to the entire system. As a result, spatial databases are organized in data structures that provide efficient access to and flexible manipulation of data. There are several ways of representing a spatial object inside a data structure [10]. Some data structures (e.g., the R-tree [7] and the Grid File [14]) 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 and by a point in a higher dimensional space in the case of the Grid File). On the other hand, another family of data structures (e.g., the quadtree [8], cell tree [6], and R^+-tree [5]) 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 family of data structures.

One problem that arises in processing queries in a relational database is duplicate elimination. Many techniques for eliminating duplicates in relational databases are addressed in the literature (e.g., [3]). Hashing (e.g., linear, dynamic, and extendible [4, 11, 12]) is one of the most common techniques for solving this problem. Although most of these techniques apply to any type of object, here we address the issue of duplicate elimination in the context of spatial objects. In particular, we attempt to take advantage of the spatial characteristics of objects as a guide to eliminating duplicates that result from spatial database operations. For example, in the case of the R^+-tree and the quadtree, duplicates arise because the object is partitioned into pieces. We can benefit from knowing that these pieces are contiguous in space (if this is really the case). In this paper we define an operation, termed *Report_Unique*, which manages to eliminate duplicates and report each object in the data structure just once, regardless of the multiplicity of the partitions of the object inside the spatial data structure.

Report_Unique can be viewed as an alternative approach to hashing when dealing with spatial data. Report_Unique maintains a dynamic data structure, termed the *active border*, that serves as a repository for the objects currently being processed, termed *active objects*. The active border data structure resembles a dynamic hash table. Using spatial properties of the underlying objects, the active border can grow and shrink in constant time. By knowing the extent and proximity of the objects, we can detect when all pieces of an object are entirely processed. At that time we can remove the object entry from the active border, thereby bounding the size of this table by the number of active objects. A look-up function, also based on spatial proximity, can be used to access entries in the active border. The look-up function is the spatial analog of a hash function. As will be demonstrated in this paper, an important advantage of proximity-based look-up functions is that as a result of their use, buckets in the active border are guaranteed not to overflow.

Another important distinction is that when conventional hashing techniques are used there is no way of predicting when an object will not be referenced again and hence can be safely deleted from the hash table. By considering the extent of spatial objects, Report_Unique is able to detect when all the pieces comprising the object have been processed and hence can delete the object. This way we can avoid the situation that the hash table grows until it reaches its maximum limit, i.e., O(number of objects in the database).

The Report_Unique operation can be used to implement a number of applications. For example, suppose we want to count the number of spatial objects in the given 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 case, we want to count each object only once regardless of the number of partitions of the object. As a specific example, suppose we want to

report the spatial objects that lie inside a window, say w. Although several partitions of the same object, 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 as is the case when 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 containing the spatial objects in question, Report_Unique ignores the multiplicity of the object partitions inside the data structure and transmits each object in the data structure to p just once. Again, our goal is to investigate the use of the spatial characteristics of objects to eliminate duplicates that result from processing a variety of spatial queries.

In answering spatial queries, parts of a spatial object may be clipped or occluded (e.g., finding the objects inside a query window). This affects the object's extent and proximity since an object may become discontinuous, have holes, etc. This is reflected in the underlying algorithm for Report_Unique as well as in its performance. We address this issue by classifying spatial objects into categories that are shown to affect the form and performance of the Report_Unique operation.

The rest of the paper is organized as follows. In Section 2 we review some duplicate elimination techniques that are commonly applied in relational database management systems. In Section 3 we outline a straightforward algorithm to solve the Report_Unique problem and analyze its worst-case space and time complexities. This algorithm resembles a simplified form of hashing. The remaining sections describe techniques to eliminate duplicates using spatial characteristics of the underlying spatial objects. Our spatial objects are line segments and rectangles. We use a PMR quadtree representation [13] for the set of line segment objects and a variant of the region quadtree [8], termed a *rectangle quadtree*, for the rectangles. The straightforward algorithms require $O(n)$ space, where n is the number of spatial objects in the database. This may be unacceptable for large databases. Section 4 describes another algorithm that eliminates the space requirements of the algorithm of Section 3. Section 5 demonstrates that supporting the Report_Unique operation for a realistic variety of objects requires classifying spatial objects into categories of different complexity. Sections 6 and 7 show how these classes affect the performance and correctness of the alternative approaches to Report_Unique. Section 8 contains some 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 the Report_Unique operation, nor do we present the most efficient algorithms for each class of the spatial objects. 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 Duplicate Elimination in Relational Databases

Duplicate elimination is an expensive operation which is frequently needed during relational query processing. For example, identical entities appear when applying the relational `PROJECT` operation. In particular, in SQL, the key word `DISTINCT` in the `SELECT` clause means that only distinct tuples should remain in the result. One way to eliminate the duplicate tuples is first to sort the entire set of entities, and then to eliminate duplicates by scanning the entire sorted list while deleting the entities which appear in consecutive positions.

Sorting is an expensive operation for a large disk file. Special *external sorting* techniques [9] are frequently used. A common method is a variation of the merge sort technique. First, the records within each block are sorted. Then sorted blocks are merged to create groups of sorted records each of size two blocks. Each such group of sorted records is sometimes called a *run*. Following

that, runs of two blocks are merged to form runs of four blocks, and so on until the final run is the completely sorted file. Assume that there are n spatial objects stored in some data structure and that on the average each object is partitioned into k pieces inside the data structure.[1] These object pieces may be a result of a previous spatial operation e.g., a spatial join. Assume further that each piece is of fixed length, and that each disk block can hold up to b pieces, where b is the blocking factor. The number of block accesses needed to sort the file that extends over m blocks (m is nk/b) is proportional to $m \log m$. The storage requirement for the sorting algorithm is m blocks.

Hashing can also we used to eliminate duplicates. As each entity is hashed and inserted into a bucket of the hash table, it is checked against those already in the bucket. If it is a duplicate, then it is not inserted into the bucket. The storage requirement for the hash table is proportional to the number of objects in the spatial data structure (i.e., $O(n)$). The hash table is checked $O(nk)$ times, i.e., once per object piece. Notice that not all the hash table buckets may fit in main memory at the same time. Therefore, $O(nk)$ bucket access requests are generated and disk I/O may be incurred to insert an object identifier into a bucket or to determine the presence of an object inside a bucket.

3 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. Report_Unique1 makes use of an array of bits. There is one bit for each object which is initially false. Report_Unique1 proceeds as follows:

1. Traverse the data structure, and for each sub-object that is 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.

Report_Unique1 also represents a simplified model of a hashing mechanism. 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. 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 requires n bits to store the bit array, and 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 the end-points of a line segment, the polygon boundary, etc.). Since all the algorithms that we present are based on traversing a list of blocks, we will omit the term C_m from the comparison. Therefore, the cost of Report_Unique1 is $knC_{cpu} + nC_{io}$. This component of the cost cannot be ignored for algorithms that are not based on traversing the data structure (e.g., direct access of data structure elements).

In order to improve the execution time or space requirements of Report_Unique1 we will make use of the fact 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 yields the value of true. Any function t that

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

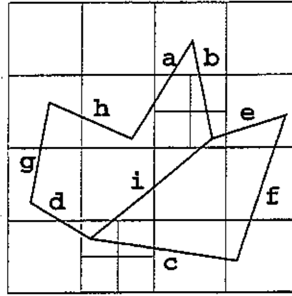

Figure 1: An example of a PMR quadtree.

satisfies this criterion can be used as a means 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 Report_Unique1, a function t_1 is defined as follows (a is the bit array and $oid(p_i)$ is the identifier of the object to which piece p_i belongs):

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

4 Avoiding the Extra Space

In order to have a concrete basis for our discussion we use a database of n line segment objects.[2] We assume that the lines are stored in a spatial data structure that partitions the line segments into pieces. In this section we present a simple reporting algorithm that does not require additional space (e.g., in contrast to the $O(n)$ space required for hashing as well as Report_Unique1). The algorithm makes use of some simple spatial characteristics of the line segment objects in order to report each line segment just once instead of as many times as it appears in the data structure.

As an example spatial data structure, we use the PMR quadtree [13]. 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.

In the PMR quadtree, a block is permitted to contain a variable number of line segments. It is constructed by inserting the line segments 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 caused 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.

Each block in the PMR quadtree stores the object identifiers of the lines passing through it. The full description of the line segments (e.g., the start and end coordinate values of the end-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

[2] A similar algorithm can be described for a database of rectangular objects.

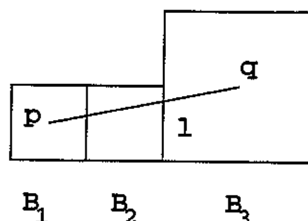

Figure 2: Line l is stored in blocks B_1, B_2, and B_3 where only block B_1 causes the point-in-block test to succeed since the starting point p of l only lies inside B_1 and not in B_2 nor in B_3.

traverses the blocks of the PMR quadtree, and for each block, say B, it performs the following actions:

1. For each object identifier 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

$$t_2(l_i, B) = if(start_point(l_i)\ in\ B)\ then\ return\ (true)\ else\ return\ (false)$$

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

Figure 2 helps in understanding the algorithm. Notice that for each line l, the testing function t_2 is true only when the block containing the starting point of l is processed. Application of t_2 to all blocks containing pieces of l other than the starting point of l yields false (t_2 is referred to as the *point-in-block test*).

A major advantage of Report_Unique2 over the test-and-set algorithm is that it does not require any additional space. However, 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. In particular, 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 Report_Unique2 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 Report_Unique2 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).[3] Notice that an algorithm for duplicate elimination based on hashing would result in the same number of bucket read operations (i.e., kn). These bucket requests may or may not result in disk I/O requests depending on the buffering techniques adopted. A better algorithm would perform only n such requests, i.e., one request per line segment in the database

[3] In order to simplify the presentation, we do not address the buffering effects which may reduce the actual disk I/O.

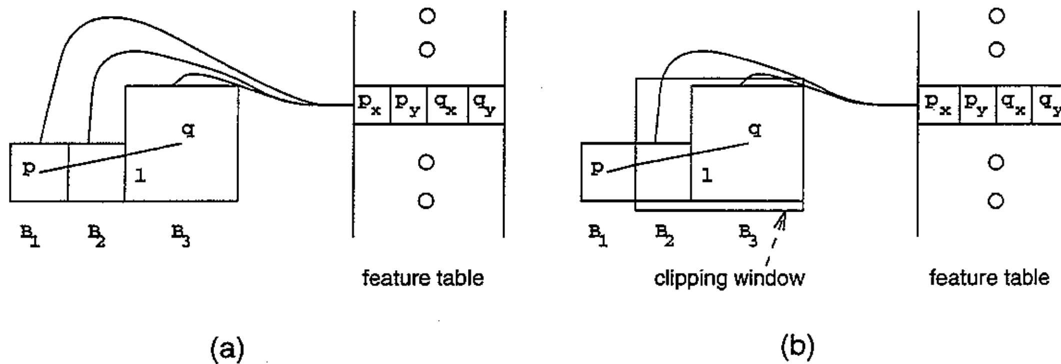

Figure 3: (a) Line l as stored in a PMR quadtree. (b) The result of clipping line l by a window operation in a PMR quadtree. Line l is partially clipped so that its original starting point in the feature table is neither in blocks B_2 nor in B_3. Notice that the feature table still stores point p as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

instead of one request for each piece of each line segment. Thus, we would like an algorithm that does not need the $O(n)$ extra storage (or at least having an asymptotic storage cost less than $O(n)$), yet one that still performs only n disk requests (e.g., n accesses to the feature table).

In Section 6, a third algorithm (Report_Unique3) is presented that overcomes this drawback. It uses the concept of an *active border* [17] to avoid accessing the feature table each time a piece of the line is encountered. However, this depends on a closer scrutiny of the nature of the spatial objects which is the subject of the next section.

5 Object Classification

An important factor affecting the performance of the Report_Unique operation is the nature 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 partially 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. This is done in order to ensure consistency when portions of line segments are added and removed as a result of set operations. The problem is that there is not enough precision to express the starting and ending points of the clipped line segments. Thus the feature table is not updated and the clipped line segment is stored implicitly. As a result, if a line, say l, in Figure 3 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. In the remaining parts of this section, we present such a classification for spatial objects of type line segment and rectangle.

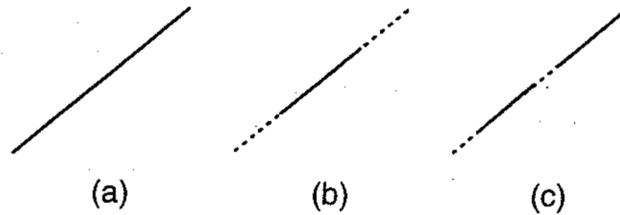

Figure 4: Classification of line segments (a) Class-1, (b) Class-2, (c) Class-3.

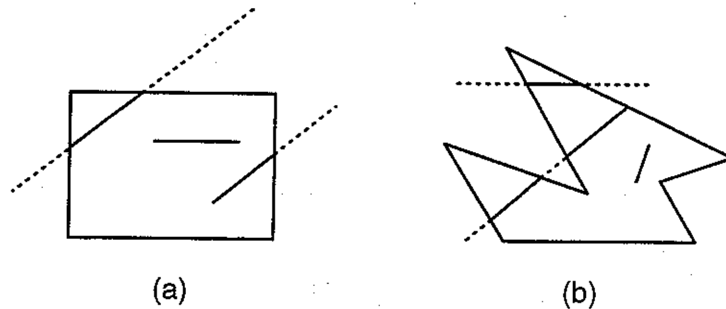

Figure 5: Example illustrating how some of the line segment classes are created: (a) Class-2, (b) Class-3.

5.1 Classification of line-segment objects

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 implementations of Report_Unique that make use of these classifications are discussed in Section 6.

- 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 (e.g., a polygon in Figure 5b). Similar classes can be constructed for other spatial objects (e.g., rectangles as illustrated in Section 5.2, or polygons).

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

- Type-1 segments: uniformly axis-parallel oriented line segments—i.e., they may be parallel to the x or the y axes but not to both (Figure 6a).
- 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 may be of an arbitrary slope (Figure 6c).

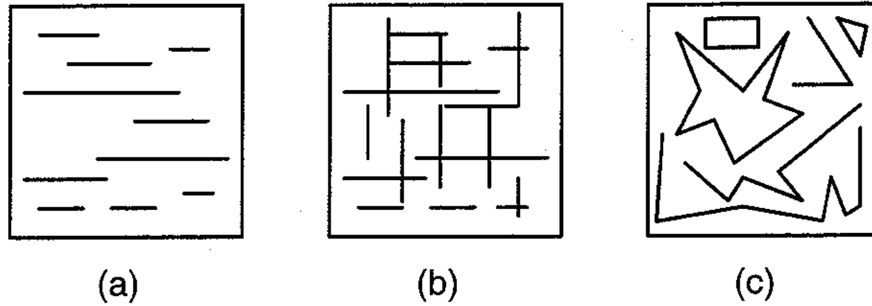

Figure 6: Three types of line segments: (a) Type-1, (b) Type-2, (c) Type-3.

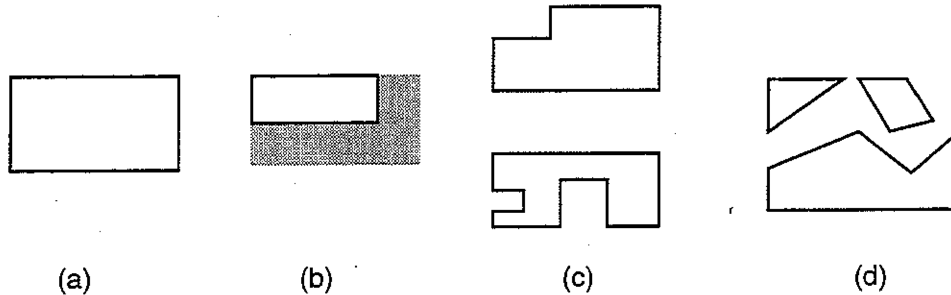

Figure 7: Classification of rectangles (a) Class-1, (b) Class-2, (c) Class-3, (d) Class-4.

5.2 Classification of rectangular objects

Spatial objects of type rectangle can be classified in the same way as line segment objects. The different classes of rectangles are given in Figure 7.

- Class-1 rectangles: a rectangle having no clipped or missing portions as illustrated in Figure 7a, and termed a *regular* rectangle.
- Class-2 rectangles: a rectangle where only a rectangular portion of it is included and the rest of the rectangle is clipped as illustrated in Figure 7b, and termed a *clipped* rectangle.
- Class-3 rectangles: a rectangle where several portions (holes) may be missing, but still the rest of the rectangle is connected, as illustrated in Figure 7c, and termed a *clipped-out* rectangle.
- Class-4 rectangles: a rectangle where several portions (holes) of arbitrary shape may be missing that result in the rectangle being decomposed into several disjoint pieces, as illustrated in Figure 7d, and termed a *disjoint* rectangle. Although disjoint, all the portions of the rectangle refer to, and represent, just one object of type rectangle.

This classification of rectangles is of practical use. Class-1 represents regular rectangles (e.g., bounding boxes of some spatial objects in a map). Class-2 represents the parts of a rectangle that remain after a rectangular window operation (Figure 8a). Class-3 represents the parts of a rectangle that lie outside one or more rectangular query windows (Figure 8b). These rectangles result from clipping-out regular rectangles against the query rectangles. Notice that each of the resulting rectangles is still representable as a four-connected region. Class-4 represents the parts of a rectangle that lie inside (or outside) one or more query windows of arbitrary shape. For example, Figure 8c is the result of intersecting a rectangle with a number of simple polygons. We further classify rectangles of each class into the following types according to their orientation:

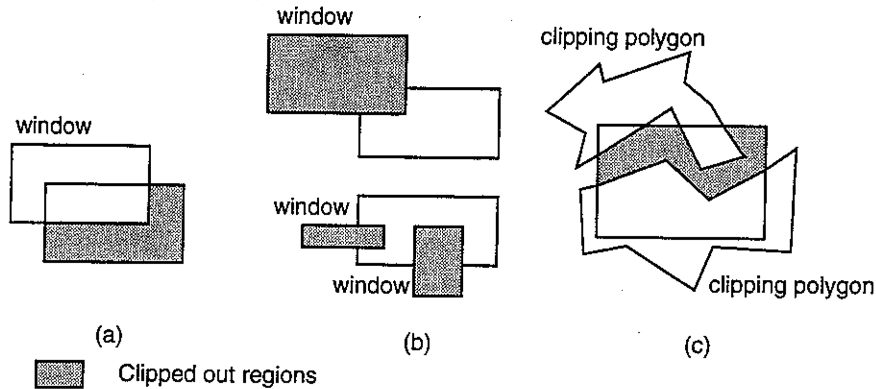

Figure 8: Example illustrating how some of the rectangle classes are created: (a) Class-2, (b) Class-3, (c) Class-4.

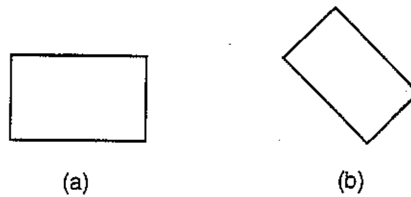

Figure 9: Two types of rectangles: (a) Type-1, (b) Type-2.

- Type-1 rectangles: rectilinear rectangles, i.e., whose sides are parallel to the axes (Figure 9a).
- Type-2 rectangles: arbitrary rectangles—i.e., rectangles whose orthogonal sides have arbitrary orientations (slopes) (Figure 9b).

6 Duplicate Elimination Algorithms for Line Segment Objects

Below, we present some implementations of algorithms for the Report_Unique problem that can handle several class/type combinations given in Section 5.1, and see how these combinations affect the complexity of the algorithms. Our underlying spatial database makes use of a PMR quadtree to store the spatial database objects which are line segments in these examples.

6.1 Class-1 Type-1 and Class-1 Type-2 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 *active set* of line segments. These are the line segments that intersect the block or pass through the block. 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 through which l passes have been visited during the traversal. The necessary storage is bounded by the maximum 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 organization of the active set depends on the operations that are to be performed on the set.

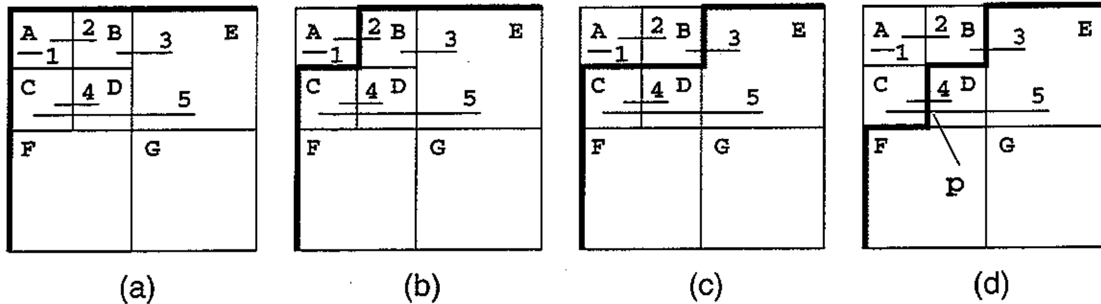

Figure 10: Various states of the active border: (a) initial condition; (b) after visiting block A, lines 1 and 2 are reported, line 1 is deleted; (c) after visiting block B, line 3 is reported, line 2 is deleted; (d) after visiting block C, lines 4 and 5 are reported.

The 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 encountered for the first time 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 algorithm visits the blocks of the quadtree in NW, NE, SW, SE scanning order. It maintains one basic data structure: an *active border* [17]. In the context of duplicate elimination and hashing, the active border is the spatial analog of a hash table. 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 horizontal edges, moving from southwest to northeast, as shown by the heavy line in Figure 10a. Initially, the active border consists of the north and west borders of 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 10b and 10c).

We define an active line segment as a line segment that is partially processed, 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 the blocks through which it passes have been processed by the traversal algorithm or if all of them are yet to be processed. There is a data bucket associated with each element of the active border. Each bucket is of capacity b, which is the same as the bucket capacity of the underlying quadtree block. Every element of the active border stores in its bucket the set of active line segments that intersect the portion of the active border corresponding to this element. For example, in Figure 10d, lines 4 and 5 are associated with portion (element) p of the active border. 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 block of an active border portion that contains the line's identifier. For example, when block B is processed in the quadtree of Figure 10c, line 2 is deleted since it is entirely covered by blocks A and B which have been processed already.

When a block in the quadtree that corresponds to a leaf node is processed, the portion of the active border that is adjacent to the block must be located. In [1], a technique is described that enables the blocks to 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

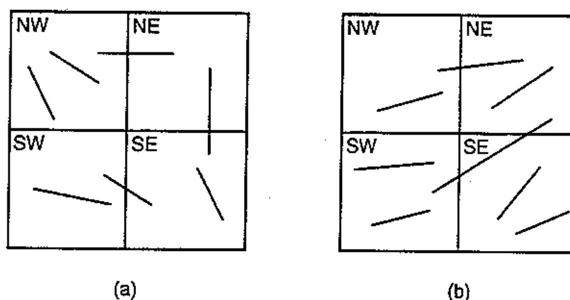

Figure 11: (a) Type-3 line segments with NW-SE orientation. (b) Type-3 line segments with NE-SW orientation.

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 this element for the existence of a line segment as well as as insertion and deletion of a line segment can each be done in $O(b)$ time which is really a constant or $O(1)$.[4]

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 bucket size) and T is the width of the space comprising the underlying spatial database. In practice, it is expected that the size of the active set is considerably smaller than $O(bT)$. Observe that it is guaranteed that the active border buckets will never overflow since there is one-to-one correspondence between elements of the active border and blocks of the underlying quadtree. Since each block can hold up to b line segments, then at worst each bucket will hold the same number of line segments and hence cannot overflow.

It is important to note that the above algorithm does not perform any disk I/O requests in order to access the feature table. The reason is that none of the steps of the algorithm require knowledge of the coordinate values of the end-points of the line segments. It suffices to know the existence of the line segment identifier in the visited block of the underlying spatial database. In addition, only the line segment identifier needs to be maintained in the active border. This is because lines are regular (Class-1) and are parallel to the axes (Type-1). As we will see in the following sections, this will no longer be true if we must be able to handle other class/type combinations. In the latter case, we will need to know the coordinate values of the end-points of the line segment and hence will need to issue I/O requests to the feature table to gain this information.

We also observe that the above algorithm works for lines that are parallel to the y axis, lines that are parallel to either axis (i.e., Type-2 objects), as well as lines of NW-SE orientations (which are a subset of Type-3 objects—see Figure 11a). As we will see in Section 6.3, line segments of NE-SW orientations (the rest of the Type-3 objects—see Figure 11b) are more difficult. However, the algorithm in Section 6.4 handles all of these cases in a uniform manner.

6.2 Class-2 and Class-3 Line Segments

The algorithm of Section 6.1 has been defined for Class-1 Type-1 line segments. It can be extended easily to support some Class-2 and Class-3 line segments. In fact, for Class-2 Type-2 objects (and

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

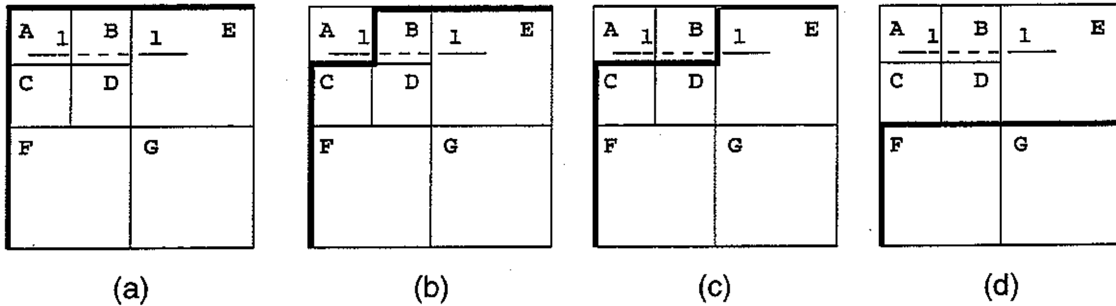

Figure 12: Example illustrating the difficulty with Class-3 Type-2 line segments. (a) initial state, (b) l is reported when block A has been processed, (c) l is deleted when block B has been processed, (d) l is reported again when block E has been processed.

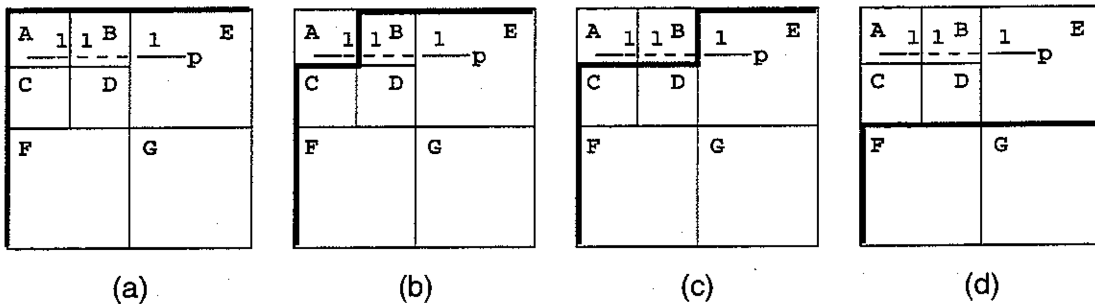

Figure 13: Class-3 Type-2 line segments. (a) initial state, (b) l is reported when block A has been processed, (c) l is still considered active when block B has been processed (although it does not overlap with B), (d) l is not reported when block E has been processed, and is deleted from E since l's end-point p lies inside E.

also Class-2 Type-1), the same algorithm applies correctly without any changes. For Class-3 Type-2 objects, a portion of a line segment can be hidden so that its absence from a neighboring block of 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 12). For this reason, the line deletion mechanism in the algorithm of Section 6.1 does not work properly. In fact, if used, it may result in a line segment being reported several times on account of its erroneous deletion from the active border.

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 in Figure 12, is encountered for the first time during the traversal, the coordinate values of l's end-points are retrieved from the feature table. The end-point, say p, of l that has not yet been encountered by the traversal is stored in the active border along with l's identifier. Now, if a neighboring quadtree block 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 or not p lies inside B. 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 13. Notice that in this case, a disk I/O request has to be issued to the feature table once per line segment in order to retrieve the coordinate values of the end-points of the line segment. However, this is only performed once for each line segment for a total of $O(n)$ disk I/O requests for the entire algorithm.

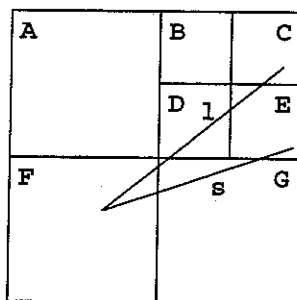

Figure 14: Example illustrating the difficulty with Class-1 Type-3 line segments.

6.3 Class-1 Type-3 and Class-2 Type-3 Line Segments

The algorithm of Section 6.2 does not work for Type-3 objects (regardless of their class). For example, consider the two lines l and s in Figure 14. This algorithm would report line segment s more than once, i.e., once for each of the visits to blocks E and F. Since block E is encountered before blocks F and G in the quadtree traversal, s will be first reported when block E is processed. When block F is encountered, s will be reported again since s's identifier does not appear in elements of the active border that are adjacent to the western and northern boundaries of F. Notice that the algorithm of Section 6.2 restricts the search of the active border in these two directions in order to guarantee constant time access to the active border. In addition, line segment l is also reported more than once—in fact, once for each of the visits to blocks C, D, and F. The problem is worse for Class-3 line segments.

In order to understand the cause of this problem we point out that it arises because of the nature of the quadtree traversal that scans the underlying space in NW, NE, SW, SE scanning order. Any line with an orientation from the NE to the SW quadrant (we term it a *NE–SW orientation*) may be reported more than once. Recall that when processing a block, say b, of the underlying quadtree, we restrict ourselves to accessing elements of the active border that are adjacent to the northern and western boundaries of b. Adopting this restriction enables us to access and update the active border in constant time. In order to report lines with a NE–SW orientation just once, one possible solution is to traverse the underlying quadtree twice, once in the NW, NE, SW, SE scanning order and a second time in the SE, SW, NE, NW scanning order [18]. Whenever a line segment identifier is encountered during the processing of a quadtree block, say b, the active border is accessed (actually, only the elements of the active border that are adjacent to the northern and western boundaries of b are accessed) to check if the line segment already exists. If the line segment is not found in the active border, we retrieve it from the feature table and compute its slope. In the second scan, we consider only the line segments with a NE–SW orientation, while in the first scan we consider all the other line segments.

The two-scan approach has two factors that adversely affect the I/O cost and hence the overall performance. First of all, the cost of the quadtree traversal process is doubled since the quadtree is traversed twice. Second, for n line segments, the feature table is accessed nk times instead of n times. The reason for the extra factor k is the need to check the orientation of all pieces of line segments that are encountered. The problem is that during the first scan, when processing a line in NE–SW orientation, each piece of the line may not be in the northern and western elements of the active border that are adjacent to the boundaries of the current quadtree block. So, in order to avoid reporting such a line, we must check its slope, which means that the line segment has to be retrieved from the feature table. Observe that only the line-segment identifier is stored in the

quadtree block. The same problem occurs with lines in NW–SE orientation during the second scan. This results in nk accesses to the feature table in the worst case.

An alternative solution to the Report_Unique problem for Class-1 Type-3 and Class-2 Type-3 line segments is achieved by adapting 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 treating the blocks through which the line segments pass as regions with appropriate connectedness conventions. In particular, the variant of the algorithm reports the coordinate values of the end-points of the line segment that passes through the region instead of the boundary. This version of the algorithm executes in $O(kn \cdot \alpha(kn))$ time where $\alpha()$ is the inverse of Ackerman's function (a function that grows very fast and hence its inverse grows very slowly). A nice feature of this algorithm is that it does not perform any disk I/O requests to the feature table.

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, thereby suggesting that a further simplification of the algorithm (from a computational complexity standpoint) may be possible. We do not address this issue here.

The above algorithm works for Class-1 Type-3 and Class-2 Type-3 line segments. In fact, the algorithm works for any type of object (not necessarily a line segment) as long as the object's pieces are connected and are not disjoint. In the following section we present a unified algorithm that works for all Class-Type combinations and that achieves good performance in almost all of the cases.

6.4 A Unified Algorithm

Our goal here is to outline an algorithm that performs relatively well for all the class/type combinations. We will concentrate on disk I/O complexity since it is the major factor in our case. The idea behind the unified algorithm is three-fold.[5]

1. We report a line l at the time l is deleted from the active border and not at the time it is inserted into the active border (Figure 15). This involves storing the bounding box of l in the active border instead of the coordinate values of the end-points of l. This is done in order to accommodate lines with NE-SW orientation. One useful property of a Z-shaped traversal for rectangular objects (e.g., the NW, NE, SW, SE order) is that by the time the block containing the lower-right corner of the rectangle is traversed, it is guaranteed that all the blocks of the underlying database that comprise the rectangle have already been traversed [2]. For bounding boxes containing line segments, regardless of the orientation of the line, once the traversal of the database blocks reaches the lower-right corner of the bounding box, we are sure that all the pieces of the line segment have already been visited during the traversal. In this case, reporting the line at this point, and then deleting it from the active border, ensures that the line will be reported just once since it will never appear again during the traversal.

2. During the traversal process, we propagate the identifier of the line segment into the active border as long as the corresponding bounding rectangle is not entirely traversed. This ensures that regardless of whether the line is regular, clipped, or broken, the line segment identifier

[5] We assume that we traverse the underlying spatial database as well as the active border simultaneously in a NW, NE, SW, SE order and that the origin is at the upper-left corner of the underlying space.

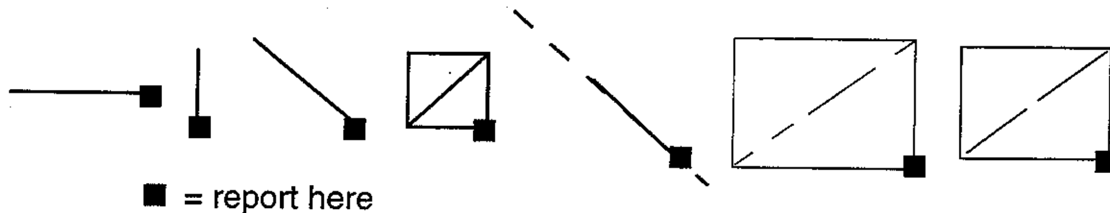

Figure 15: The algorithm reports a line segment after the line segment is entirely spanned by the active border.

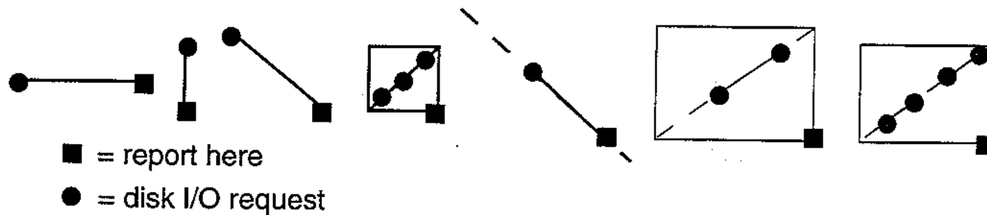

Figure 16: A disk I/O request is issued to access the feature table when the line segment identifier is encountered in the traversal and is not found in the neighboring element of the active border. Notice that lines with NE-SW orientations may issue redundant disk I/O requests.

will always be propagated in the active border until all the pieces of the line segment have been processed.

3. We relax the restriction that we perform only one disk I/O request to the feature table per line segment. Although more than one access to the same line-segment in the feature table is redundant, this will help in providing a new algorithm for the Class/Type combinations that were not covered in the previous sections, mainly clipped or broken lines with arbitrary orientation, i.e., Class-2 Type-3 and Class-3 Type-3 line segments. In addition, the algorithm will still perform one disk I/O request to the feature table per line segment for the simpler classes (in other words we will not incur any increases in complexity because of adopting this unified approach).

The algorithm (termed *Unified1*) proceeds as follows:

1. Traverse the underlying spatial database as well as the active border simultaneously in the NW, NE, SW, SE order.
2. Maintain in the active border:
    (a) the identifiers of the lines that intersect the active border or whose pieces have not yet been entirely processed by the algorithm, and
    (b) the bounding box of each line described above.
3. Report the identifier of each line l when the block containing the lower-right corner of l's bounding box is reached, and delete l's information from the active border.

Figure 16 illustrates the situation which arises when a disk I/O is requested for each Class/Type combination given in Figure 15. Notice that only for the case of lines with a NE-SW orientation will there be more than one disk I/O request. Figure 17 illustrates the worst case number of disk

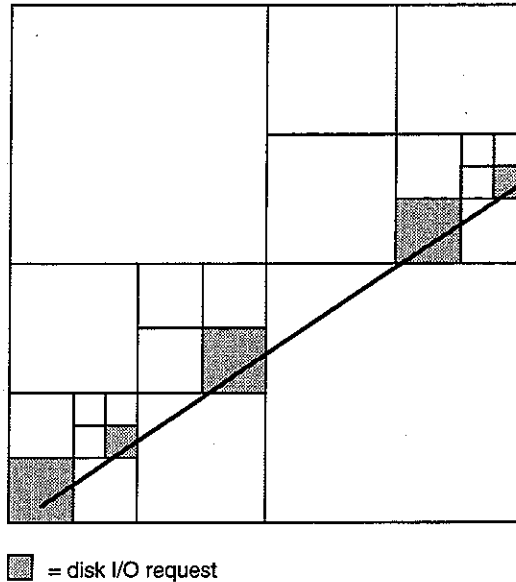

Figure 17: The maximum number of disk I/O requests that can arise for a line segment with NE-SW orientation.

I/O requests that are required with lines of this orientation. Observe that a disk I/O request occurs with every other piece in the object. Therefore, in the worst case $\lceil \frac{k}{2} \rceil$ disk I/O requests are issued. The unified algorithm performs a disk I/O request even for the case of Class-1 Type-1 objects. In order to avoid this, we add the following two exceptions at the beginning of the algorithm, resulting in this final version of the algorithm (termed *Unified2*) given below:

1. If all the line segments in the underlying spatial database are regular (Class-1) and are either parallel to the x axis, or parallel to the y axis, or are of NW-SE orientation, then perform the algorithm of Section 6.1.
2. Else if the line segments in the underlying spatial database are regular or broken (Class-1 and Class-2) and have arbitrary orientation (Type-3), then perform the algorithm of Section 6.3.
3. Else perform the algorithm Unified1.

Letting n_{NW-SE} and n_{NE-SW} be the number of lines with a NW-SE and NE-SW orientation, respectively, we have the following disk I/O complexity for algorithm Unified2:

- No disk I/O requests for regular lines (Class-1) that are either parallel to the x axis, or to the y axis, or that are of an arbitrary orientation.
- n disk I/O requests for clipped and broken lines (Class-2 and Class-3) that are parallel to the x axis, or to the y axis, or that have a NW–SE orientation.
- $\lceil \frac{k}{2} \rceil n_{NE-SW}$ disk I/O requests for clipped or broken lines (Class-2 and Class-3) that are of a NE–SW orientation.

The complexity of Unified2 is similar to the other algorithms presented in Sections 6.1–6.3 in addition to the fact that it handles broken lines having a NE-SW orientation for which no corresponding algorithm is given in Section 6.

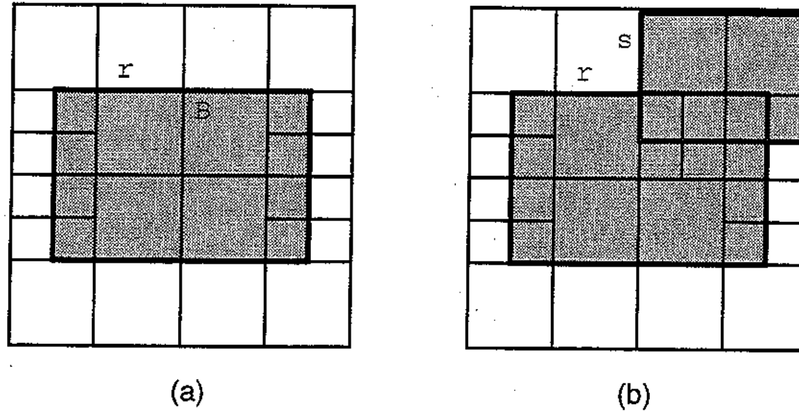

Figure 18: (a) The decomposition of rectangle r into its constituent quadtree blocks. (b) Block B is decomposed after rectangle s is inserted.

7 Duplicate Elimination Algorithms for Rectangular Objects

Below, we present some implementations of algorithms for the Report_Unique problem that can handle several class/type combinations of rectangular objects given in Section 5.2, and see how these combinations affect the complexity of the algorithms. The rectangles are stored in a variant of the region quadtree [8] which we term a *rectangle quadtree.* In particular, each rectangle is decomposed into the quadtree blocks that lie inside the rectangle. For example, Figure 18a shows the decomposition of rectangle r into its constituent quadtree blocks. We assume that the rectangles need not be disjoint. We also adopt the restriction that each quadtree block is completely inside all of the rectangles that overlap it. In other words, the case that only part of a quadtree block lies in a rectangle is not allowed. For example, this restriction means that block B of Figure 18a must be decomposed when rectangle s is added to yield the decomposition given in Figure 18b.

Our representation assumes that all the blocks that are internal to a rectangle store an object identifier which is an offset in a feature table where the upper-left and bottom-right coordinate values of the rectangle are stored. As in the case of line segments, when a rectangle is clipped (e.g., as a result of a window operation), the rectangle quadtree does not update the coordinate values of the the clipped rectangle in the feature table. Notice that we cannot simply traverse the feature table and report the rectangles that we find since not all the rectangles in the feature table necessarily belong to the data structure in question.

It is important to observe that rectangles of the underlying spatial database are allowed to overlap arbitrarily in space. Let q be the maximum number of overlapping rectangles at any point in space, i.e., for any block, there are at most q overlapping rectangles. As we will see in the following section, q is also the maximum number of entries that can be stored at any given element in the active border, i.e., q plays the same role as the bucket size b in the PMR quadtree described in Section 4.

7.1 Class-1 Type-1 and Class-2 Type-1 Rectangles

Assume a collection of rectangles whose sides are parallel to the x and y axes where some of the rectangles can be clipped (Class-2) but are still of rectangular form. The algorithm traverses the rectangle quadtree block-by-block and use the active border to maintain the *active set* of rectangles.

Our goal for this Class/Type combination is not to perform any I/O to the feature table. In this case, the only information at hand are the rectangle identifiers that are stored in the quadtree blocks. We now show that this information is sufficient for this Class/Type combinations.

Rectangle identifiers are added to the active border when they are encountered during the traversal, and propagated into the active border until all the pieces of each rectangle have been visited. A rectangle identifier is reported at the time it is being deleted from the active border. The problem is how to detect that a rectangle has been processed in its entirety when there is no means for the algorithm to know the coordinate values of the end-points of the rectangle. In order to capture this information, the algorithm needs to maintain some temporary information in the active border to help it determine that the lower-right corner of the rectangle has been reached and that the rectangle identifier can be reported.

For example, consider the rectangle r given in Figure 18a. We use Figure 19 to illustrate the process of reporting its presence. Heavy shading is used to indicate that the block has already been traversed. When block A is encountered during the traversal (Figure 19a), r's identifier (associated with A) is inserted into the active border. When block B is visited (Figure 19b), r's identifier is propagated into the active border. Upon encountering a western or southern boundary of a rectangle for the first time (e.g., when processing block C in Figure 19c) a special marker symbol, say e_r, is associated with the active border element that led to the detection of this fact (e.g., adjacent to block C). Notice that e_r is propagated into the active border when block D is processed (Figure 19d). As another example, we observe that a special marker symbol, say s_r is added to the active border element contiguous to block E (Figure 19e) and is propagated into the active border elements to the east of E, when their adjacent blocks in the database are processed. At the time block F is processed (Figure 19f), the active border elements to the north and west of F will contain the special markers e_r and s_r, respectively. This situation serves to indicate that we are through. At this point, both e_r and s_r are deleted from the active border and r's identifier is reported.

Notice that r is reported once all the blocks comprising r have been visited by the traversal procedure. For example, in Figure 19f, at the time block F is visited, we are sure that all the blocks inside r are already processed by the algorithm. This is because a NW, NE, SW, SE traversal order is admissible [2].

As demonstrated here, our algorithm does not need to perform any disk I/O requests to access the feature table. The time complexity of this algorithm is $O(nkq)$ which is proportional to the number of object pieces in the database that are encountered during the traversal. Notice that the factor q is included because all the procedures that access the active border perform in $O(q)$ time, as described in the introduction of Section 7. The space complexity is $O(\textit{active rectangles})$, where the active rectangles are the ones that intersect the active border at any given point. The worst case size of this set is $O(qT)$, as shown in Section 6.1. Notice that the algorithm works correctly even if rectangles are clipped due to some intersection with a rectangular window (i.e., rectangles of Class-2) as it does not depend on the actual coordinate values of the rectangle since it does not access the rectangle entry in the feature table.

7.2 Class-3 Type-1 and Class-4 Type-1 Rectangles

Class-3 and Class-4 imply that the rectangles may have notches on them or may even consist of disjoint pieces. For these classes of rectangles, we cannot avoid accessing their entries in the feature table in order to know the real extent of the rectangles while traversing them. Rectangle identifiers will be propagated in the active border (regardless of whether we are traversing a part of the rectangle or a clipped out portion of the rectangle) until the lower-right corner of the rectangle

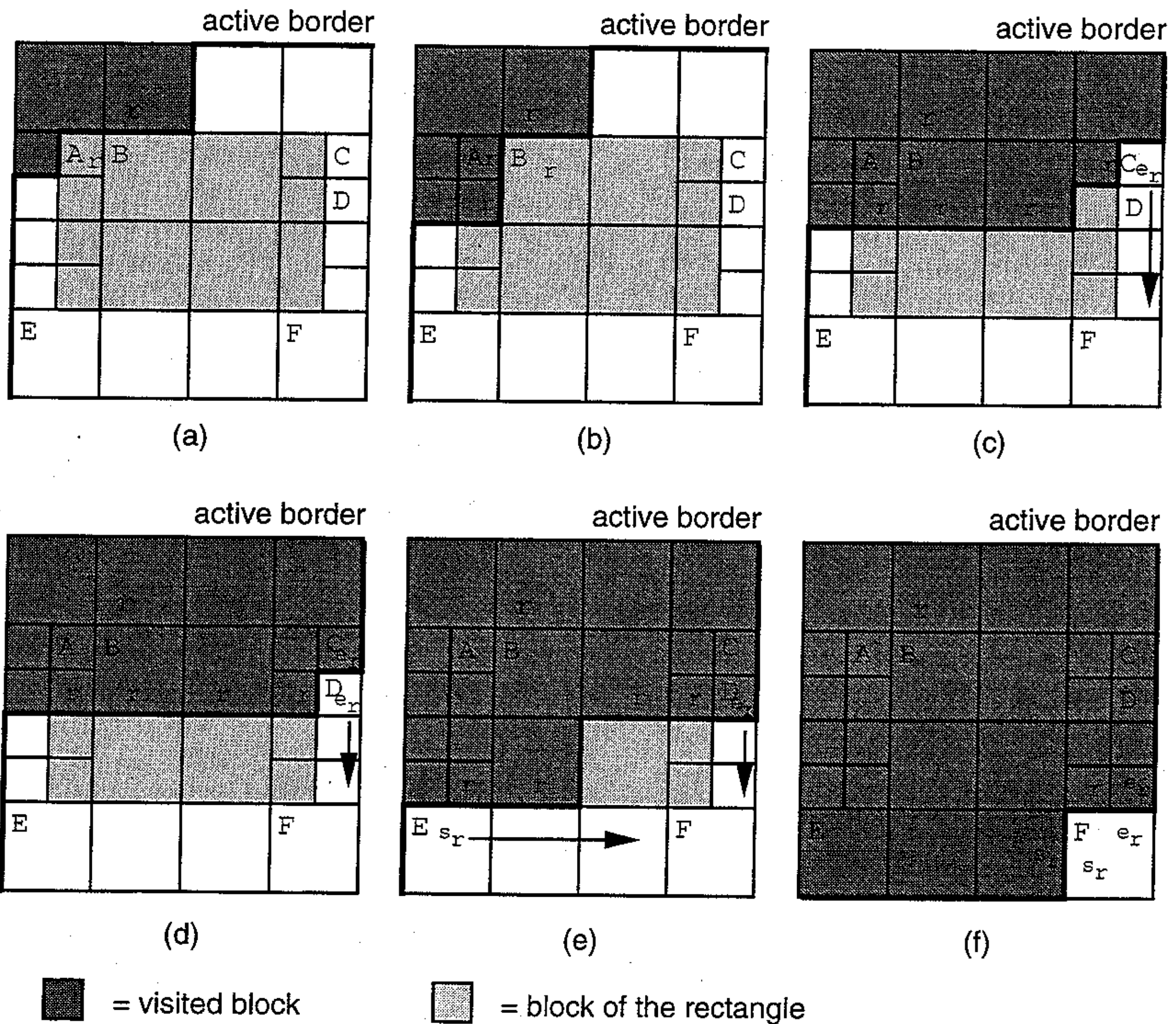

Figure 19: (a) When block A is processed r is inserted into the active border. (b) r is propagated into the active border when block B is processed. (c) When block C is processed a special marker symbol e_r is inserted into the active border. (d) e_r is propagated into the active border when block D is processed. (e) When block E is processed a special marker symbol s_r is inserted into the active border. (f) r is reported at the time block F is processed and both e_r and s_r are deleted from the active border.

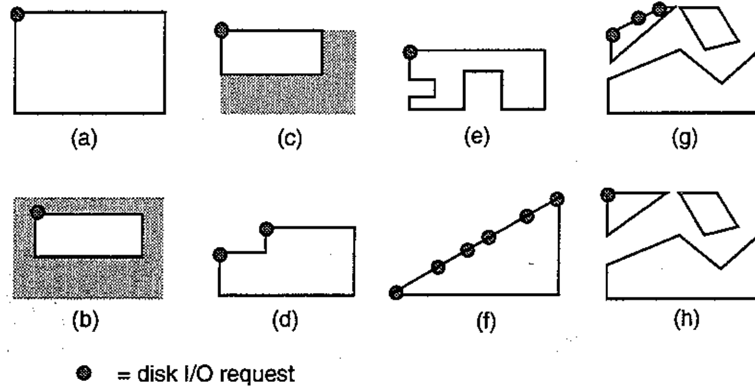

Figure 20: (a)–(c) When the rectangle is first encountered a disk I/O request is issued to retrieve the rectangle's coordinate values. (d) Two disk I/O requests are issued whenever the upper-left corner of the rectangle is clipped out (Class-3). (e) Although the rectangle is clipped from several locations, only one disk I/O request is issued since the upper-left corner of the rectangle is not clipped out (Class-3). (f) The worst-case disk I/O requests for Class-3 rectangles occurs when many query rectangles clip out portions of the rectangle such that the upper-left piece of the remaining portion of the rectangle forms a line segment of NE-SW orientation. (g) Redundant disk I/O requests are issued when the upper-left piece of the rectangle is processed since the upper-left corner of the rectangle is clipped out (Class-4). (h) Only one disk I/O request is issued although the rectangle is partitioned into more than one disjoint piece (Class-4). Notice that the upper-left corner of the rectangle is not clipped out.

is visited by the traversal. In this case, the rectangle is reported by the algorithm and deleted from the active border. This implies that the algorithm has to issue some disk I/O requests to the feature table. The algorithm proceeds as follows:

1. Traverse the underlying spatial database as well as the active border simultaneously in the NW, NE, SW, SE order.
2. Maintain in the active border:
   (a) the identifiers of the rectangles that intersect the active border or whose pieces have not yet been processed in their entirety by the algorithm, and
   (b) the coordinate values of the end-points of the rectangles in (a).
3. When the block containing the lower-right corner of rectangle r is reached, report the identifier of r, and delete r's information from the active border.

Figure 20 illustrates the locations at which a disk I/O request is issued and when a rectangle is reported for several possible rectangles in Class-3 and Class-4. Notice that for Class-3 rectangles, one or more disk I/O requests can be issued. For example Figures 20a–c and e have only one disk I/O request, Figure 20d has two disk I/O requests as the upper-left corner of the rectangle is clipped-out, while Figure 20f has many disk I/O requests. For Class-4 rectangles, more than one disk I/O request is possible (e.g., Figure 20g) depending on the number of blocks in one of the four line segments of the window boundary that has a NE-SW orientation and is closest to the origin of the underlying space. Notice that for Class-4 rectangles, if the upper-left corner of the rectangle is not clipped out, then only one disk I/O request is issued regardless of the number of disjoint pieces

of the rectangle (e.g., Figure 20h). Since the upper-left corner of the rectangle is processed first, a disk I/O request is issued to the feature table and the coordinate values of the end-points of the rectangle are made known to the active border. This helps avoid any further disk I/O requests. Observe that this is not the case when the upper-left corner of the rectangle is clipped out (e.g., Figure 20f).

Figures 20f and 20g illustrate the cases where the maximum number of disk I/O requests for Class-3 and Class-4 rectangles can be generated, respectively. Let k_{width} be the number of blocks in the edge of the window with a NE–SW orientation that is closest to the origin of the underlying space. Then, the maximum number of disk I/O requests to the feature table for Class-3 and Class-4 is $\lceil \frac{k_{width}}{2} \rceil$. This is similar to the case of line segments with a NE–SW orientation, illustrated in Figure 17.

The algorithm described above is similar to algorithm Unified2 of Section 6.4. Therefore, in order to make it work for all classes and still avoid performing disk I/O requests for Classes 1 and 2, we check for these cases explicitly at the beginning of the algorithm.

It is worth mentioning that the task of uniquely reporting spatial objects is different from that of connected component labeling [15, 16]. In the latter task we are interested in assigning a unique identifier to each four- (or eight-) connected region in the underlying space. Although in Section 6.3 we were able to adapt one of the connected component labeling algorithms ([1]) to perform as Report_Unique, this is not generally the case. The principal difference is that, generally speaking, the inputs of the two tasks are different. More specifically, in Report_Unique we can always perform a disk I/O request to the feature table and retrieve the object's exact description. This information is not made available to the connected component labeling algorithms. In fact, the main purpose of the connected component labeling algorithms is to build the object's exact description from the local information given in the underlying space.

As a result of the difference in the nature of the type of input and the goals of the two tasks, their complexities are different. For example, Figure 21 demonstrates the worst-case complexity for the connected component labeling algorithm (measured in terms of the new labels that have to be introduced). On the other hand, the shape of the region in Figure 21 does not represent the worst case disk I/O complexity for Report_Unique. In particular, only two disk I/O requests will be issued (i.e., when blocks A and B are visited). The worst case number of disk I/O requests for Report_Unique for clipped-out rectangles (Class-3) is given in Figure 20f. Notice that having a saw-tooth-like shape (e.g., as in Figure 21) does not change the worst case since at block B of Figure 21, a disk I/O request is issued to retrieve the exact description of the rectangle from the feature table, and this description will be propagated in the active border in the eastern and southern directions. This will cover all of the saw-teeth to the east and south of block B, thereby obviating the need for additional disk I/O requests (e.g., at block C).

7.3 Rectangles of Type-2

In order to ensure that rectangles of arbitrary orientations are reported only once, we maintain their rectilinear bounding box in the active border (Figure 22a) and report the rectangle identifier once the lower-right corner of the bounding box is covered by the traversal. This is similar to the approach given in Section 6.4 when dealing with lines of arbitrary orientations. The number of disk I/O requests to the feature table is proportional to $\lceil \frac{k_{width}}{2} \rceil$, where k_{width} is the number of blocks in the line segment of the window boundary with a NE-SW orientation that is closest to the origin of the underlying space (Figure 22b). Notice that once this set of disk I/O requests is issued, the algorithm does not need to access the feature table again since the propagation of the rectangle

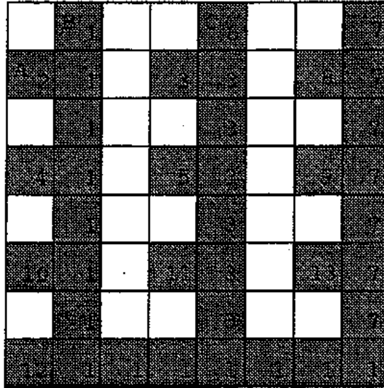

Figure 21: Sample image of size 8×8, which results in the generation of a maximum number of equivalence pairs for the connected component labeling algorithm.

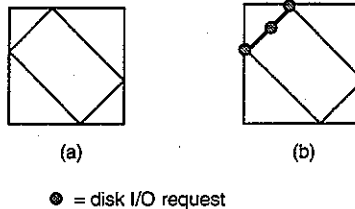

Figure 22: (a) A bounding box is stored in the active border for rectangles of arbitrary orientations (Type-2). (b) Several disk I/O requests occur when processing the line-segment closest to the origin (the upper-left corner of the underlying space).

identifier and coordinate values in the active border will help avoid any further disk I/O requests to the feature table.

8 Conclusion and Future Research

Tables 1 and 2 represent a summary of our results. They show the different complexities of the algorithms developed for the class/type object combinations that we explored. As we can see, duplicate elimination using the spatial characteristics of objects is a challenging problem. In particular, using the spatial characteristics of the objects enables us to adapt techniques used with hashing. For example, we were able to reduce the number of disk I/O requests (e.g., in comparison to $O(nk)$ disk I/O requests when using hashing), sometimes to $O(n)$ or even 0 which is a significant improvement. In addition, by using the extent and proximity of spatial objects we were able to detect when an object is no longer needed in the active border data structure (analogous to a hash table), and hence we were able to bound the size of the table. Future work involves developing better algorithms for other class/type combinations, classifying spatial objects besides line segments and rectangles in an analogous manner as well as developing algorithms for them, and considering other spatial data structures that partition objects besides the quadtree variants.

Table 1: Summary of time and space worst-case complexity of Report_Unique for class/type combinations of line segment objects. $\alpha()$ is the inverse of Ackerman's function.

	Type-1	Type-2	Type-3
Class-1	$O(kn)$ cpu, no i/o $O(T)$ space, O(active objects) avg. space	$O(kn)$ cpu, no i/o $O(T)$ space O(active objects) avg. space	$O(kn \cdot \alpha(kn))$ cpu, no i/o $O(T)$ space O(active objects) avg. space
Class-2	$O(kn)$ cpu, no i/o $O(T)$ space O(active objects) avg. space	$O(kn)$ cpu, no i/o $O(T)$ space O(active objects) avg. space	$O(kn \cdot \alpha(kn))$ cpu, no i/o $O(T)$ space O(active objects) avg. space
Class-3	$O(kn)$ cpu, n i/o $O(T)$ space O(active objects) avg. space	$O(kn)$ cpu, n i/o $O(T)$ space O(active objects) avg. space	$O(kn)$ cpu, $n_{NW-SE} + \lceil \frac{k}{2} \rceil n_{NE-SW}$ i/o $O(T)$ space O(active objects) avg. space

Table 2: Summary of time and space worst-case complexity of Report_Unique for class/type combinations of rectangle objects. $\alpha()$ is the inverse of Ackerman's function.

	Type-1	Type-2
Class-1	$O(knq)$ cpu, no i/o $O(qT)$ space, O(active objects) avg. space	$O(knq)$ cpu, $\lceil \frac{k_{width}}{2} \rceil n$ i/o $O(qT)$ space O(active objects) avg. space
Class-2	$O(knq)$ cpu, no i/o $O(qT)$ space, O(active objects) avg. space	$O(knq)$ cpu, $\lceil \frac{k_{width}}{2} \rceil n$ i/o $O(qT)$ space O(active objects) avg. space
Class-3	$O(knq)$ cpu, $\lceil \frac{k_{width}}{2} \rceil n$ i/o $O(qT)$ space O(active objects) avg. space	$O(knq)$ cpu, $\lceil \frac{k_{width}}{2} \rceil n$ i/o $O(qT)$ space O(active objects) avg. space
Class-4	$O(knq)$ cpu, $\lceil \frac{k_{width}}{2} \rceil n$ i/o $O(qT)$ space O(active objects) avg. space	$O(knq)$ cpu, $\lceil \frac{k_{width}}{2} \rceil n$ i/o $O(qT)$ space O(active objects) avg. space

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] M.B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, April 1992.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 1989.

[4] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[5] 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, CA, May 1987.

[6] 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, CA, February 1989.

[7] 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, MA, June 1984.

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

[9] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1978.

[10] 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.

[11] P. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, April 1988.

[12] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, pages 212–223, Montreal, Quebec, Canada, October 1980.

[13] 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, TX, August 1986).

[14] 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.

[15] A. Rosenfeld and A. C. Kak. *Digital Picture Processing.* Academic Press, New York, second edition, 1982.

[16] H. Samet. Connected component labeling using quadtrees. *Journal of the ACM*, 28(3):487–501, July 1981.

[17] 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.

[18] J. Snoeyink. Personal communication, August 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
May 1993

3. REPORT TYPE AND DATES COVERED
TECHNICAL

4. TITLE AND SUBTITLE
DUPLICATE ELIMINATION USING PROXIMITY IN SPATIAL DATABASES

5. FUNDING NUMBERS
DAAB10-91-C-0175

6. AUTHOR(S)
Walid G. Aref and Hanan Samet

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Center for Automation Research
University of Maryland
College Park, MD 20742-3275

8. PERFORMING ORGANIZATION REPORT NUMBER
CAR-TR-671
CS-TR-3067

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army, AMSEL-RD-IEW-TRF, Vint Hill Farms Station
Warrenton, VA 22186-5200
Office of Naval Research, 101 Marietta Tower
Atlanta, GA 30303

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A spatial database allows spatial objects to be indexed. In a spatial database, an object may extend arbitrarily in space. As a result, many spatial data structures (e.g., the quadtree, the cell tree, the R^{+}-tree) represent an object by partitioning it into multiple, simple pieces, each of which is stored separately inside the data structure. Many operations on these data structures are likely to produce duplicate results because of the multiplicity of object pieces. A novel approach for eliminating duplicate results based on proximity of spatial objects is presented. 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 indexed by the underlying representation. Example algorithms are rpesetned that perform Report_Unique for a quadtree representation of line segments and for the more general case of arbitrary rectangles. The complexity of the Report_Unique operation is seen to depend on a geometric classification of different instances of the spatial objects. Such classifications are presented for spatial objects consisting of line segments, as well as rectangles.

14. SUBJECT TERMS
spatial databases, design and analysis of algorithms duplicate elimination, data structures, quadtrees

15. NUMBER OF PAGES
30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C -	Contract	**PR** -	Project
G -	Grant	**TA** -	Task
PE -	Program Element	**WU** -	Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.