1990

Efficient geographic information systems: Data structures, Boolean operations and concurrency control

James Sheng Min

Efficient Geographic Information Systems: Data Structures, Boolean Operations and Concurrency Control

by

James Min Sheng

A thesis submitted to
The Faculty of the Computer Science Department,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.
Rochester Institute of Technology

1990

Rochester Institute of Technology
Computer Science Department

Efficient Geographic Information Systems: Data Structures, Boolean Operations and Concurrency Control

by

James Min Sheng

A thesis submitted to
The Faculty of the Computer Science Department,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by: 5 July 90

6/20/90

July 5, 90

To my wife, Olivia, and daughter, Isabel,

my parents

and

my parent-in-laws, Mr. and Mrs. Liu

Acknowledgment

I wish to express my gratitude to my thesis committee chairman Professor Peter Anderson, for his encouragement and patience. Many thanks also are due to my thesis committee member, Professor Henry Etlinger, for his helpful input.

I wish to thank the Computer Science Department, Rochester Institute of Technology for providing me the opportunity to complete my Master's Degree and Dr. Minder Chen, a visiting professor at the Management Information Systems Department, University of Arizona, for his support and friendship.

Special thanks to my mother-in-law, Mrs. Liu, for her constant encouragement and the love and care of my family during the past year.

Abstract

Geographic Information Systems (GIS) are crucial to the ability of governmental agencies and business to record, manage and analyze geographic data efficiently. They provide methods of analysis and simulation on geographic data that were previously infeasible using traditional hardcopy maps. Creation of realistic 3-D sceneries by overlaying satellite imagery over digital elevation models (DEM) was not possible using paper maps. Determination of suitable areas for construction that would have the fewest environmental impacts once required manual tracing of different map sets on mylar sheets; now it can be done in real time by GIS.

Geographic information processing has significant space and time requirements. This thesis concentrates on techniques which can make existing GIS more efficient by considering these issues:

- Data Structure,
- Boolean Operations on Geographic Data,
- Concurrency Control.

Geographic data span multiple dimensions and consist of geometric shapes such as points, lines, and areas, which cannot be efficiently handled using a traditional one-dimensional data structure. We therefore first survey spatial data structures for geographic data and then show how a spatial data structure called an R-tree can be used to augment the performance of many existing GIS.

Boolean operations on geographic data are fundamental to the spatial analysis common in geographic data processing. They allow the user to analyze geographic data by using operators such as AND, OR, NOT on geographic objects. An example of a boolean operation query would be, "Find all regions that have low elevation AND soil type clay." Boolean operations require significant time to process. We present a generalized solution that could significantly improve the time performance of evaluating complex boolean operation queries.

Concurrency control on spatial data structures for geographic data processing is becoming more critical as the size and resolution of geographic databases increase. We present algorithms to enable concurrent access to R-tree spatial data structures so that efficient sharing of geographic data can occur in a multi-user GIS environment.

Contents

List of Figures

1 Introduction and Background

1.1 Problem Statement

Geographic Information Systems (GIS) are crucial to the ability of governmental agencies and businesses to record, manage, and analyze geographic data efficiently. Installed systems number in the thousands, and by 1990 their number is expected to double [8]. This thesis will examine some of the important issues in Geographic Information System design and their effects on performance and will present strategies to improve future GIS performance.

Traditionally, geographic data have been stored on paper maps, but processing of paper maps is very labor intensive and time consuming. Storing maps on a computer makes performing statistical and quantitative analysis on geographic data much easier and faster. It also minimizes the errors that can occur, thus making the final results more accurate. Demonstration of these advantages has given rise to the widespread adaptation of GIS.

The three areas in GIS design to be addressed in this thesis are:

- Data structure,
- Boolean operations on geographical data,
- Concurrency control.

Geographic data processing differs from traditional one-dimensional numer-

ical data processing in that geographic data are spatial and multidimensional. Geographic data are usually shapes such as points, lines, and areas that represent road intersections, road segments, or forest boundaries. To process this type of data efficiently requires data structures that are specially designed for spatial data. Many structures such as quadtrees [28, 46], and K-d trees [6] have been proposed for management of geographic data. Burroughs [8] and Samet [46] have surveyed some of the data structures used in GIS.

The choice of a data representation scheme for use in a GIS can affect system performance. In this thesis, a critical survey of data representation schemes useful in geographical data processing will be conducted first. Preliminary findings in this area show that the "R-tree" data structure [27] shows promise for good management of large geographic databases.

Boolean (i.e., set-theoretic) operations on geographic data are very important in geographic data processing. They allow the user to identify regions that have a set of specific attributes. For example, the user may wish to determine regions that have low elevation and soil ph level below 7.0. To identify these regions it is necessary to intersect all low elevation regions with regions that have ph level below 7.0. Each of the resulting regions would then have the attributes of both low elevation and ph level below 7.0. The problem with boolean operations is that they require large amounts of computation time. This can become a major bottleneck when the user tries to find regions with a large number of attributes. Attempts at improving the efficiency of boolean operations have led to the adaptation of new data structures (such as quadtrees) and development

of new algorithms (such as the Weiler-Atherton polygon intersection algorithm [62]). However, no good solution has been found that will enable fast boolean operations to be made independently of the underlying data representation. This thesis will present a generalized solution for boolean operations that will not be affected by the data representation scheme used.

Concurrency control is important in the management of large databases. Currently, no concurrent algorithm exists for many of the spatial data structures used in GIS implementations. We will describe a concurrent algorithm developed especially for the R-tree data structure.

1.2 Previous Work

The development of Geographic Information Systems is relatively recent, having begun in the 1960s with the development of various programs for map display, but it was not until the late 1970s that systems were advanced enough to resemble today's GIS.

Many of the advances in GIS were driven by advances in other disciplines such as image processing and computer graphics. Data structures such as quadtrees were initially developed for computer graphics applications [28]. K-d trees were developed for searches on composite keys [6]. Although polygonal data structures are very similar to the structures used in the CORE and GKS computer graphics standards, geographic data processing had special requirements that were different from those of computer graphics and other disciplines.

A geographic database contains enormous quantities of data that usually reside on secondary storage devices. This means the data structure for implementation of a GIS must minimize disk accesses and support applications common to geographic data processing. An R-tree is a data structure that was developed for CAD and geographic data processing. Like the B-tree, it is a height balanced tree that minimizes disk access. It also supports efficient range searches in multiple dimensions. Preliminary research indicates that it has many unique advantages over other structures.

Boolean operations are a very important and common form of processing on geographic data. Most literature on boolean operations relates to a particular type of data structure. Hunter and Steiglitz [28] present algorithms for performing boolean operations on quadtrees. Boolean operations on polygons are shown by Weiler and Atherton [62]. However, no general solution for efficient boolean operations exists. The performance of boolean operations under any known data structure is very dependent on the actual data. A solution must be found that is minimally affected by the actual data stored and their representation. Optimization techniques [3] have been proposed to improve time performance of program codes. There are similarities between a boolean operation query and code blocks. They both have a structured syntax, i.e., ($\times$, +, ...etc. for arithmetic expressions and $\cap$, $\cup$, etc. for boolean operations), and both can be used to compose complex expressions. These similarities suggest that it is possible to devise a strategy to optimize a boolean operation query along the lines of compiler optimization. A literature search in this area as well as experience working with GIS, indicates, however, that no query optimization

strategy currently exists for handling boolean operations queries.

The size of geographic databases is increasing as remote sensing technology brings in data of extremely high resolution. The amount of storage required makes it no longer feasible to make multiple copies of sections of a geographic database. Concurrency control for geographic databases is increasing in importance as more users share usage of a centrally located geographic database. There are few descriptions in the literature of concurrency control of data structures for geographic data processing. Chien and Kanade [14] present techniques for distributed processing on quadtrees, but the emphasis is on parallel processing rather than concurrent access. The scarcity of concurrency control on data structures for geographic data processing can be attributed to these factors:

- Data structures for geographic data processing have only been developed recently.
- There is no consensus on the "best" data structure for geographic data processing.
- Until recently the need for concurrent access did not exist because systems could support only one user at a time.

2 Survey of Spatial Data Structures for Geographic Information Systems

In this section we perform a general survey of data structures for representing geographic data that have been proposed by various practitioners representing geographic data. These are data structures that have had a significant impact in the field of geographic information processing and are generally accepted to be effective representations for geographic data. Relative weight is given to those data structures which have been studied in more detail by many sources.

The potential for each data structure is accounted for. For example, the grid file data structure has not received much coverage in the literature. However, because they address the important issue of disk accessing in geographic data management, we shall present them here. We have divided our presentation based on temporal and structural factors and include earlier structures such as cellular and vectors, as well as more recent structures such as R-trees and cell trees.

2.1 Earliest data structures

This section describes two of the earliest types of representations used for geographic data, cellular and vector data structures. Both cellular and vector data structures are primitive in the sense that they represent the data without ordering them or exploiting any of their spatial properties in order to obtain improvements

in space and time. Deficiencies inherent in these two data structures have led to the development of advanced data structures such as quadtrees and k-d trees to improve space utilization (as in the quadtree) or searching performance (as in the k-d tree).

2.1.1 Cellular Data Structure

One of the earliest and simplest data structures used to represent geographic data is the cellular data structure. In a cellular representation, geographic space is viewed as a flat cartesian surface. The area of coverage is divided equally into an array of cells. The cells in the array are of uniform size and represent a square area of the world. The cellular data structure is used extensively in other areas of study including computer graphics, image processing, remote sensing and others. The cellular structure is also called a raster structure; its cells are called pixels.

Due to its structure simplicity, a cellular data structure can be handled easily by most software. It can be stored in an array and its contents can be referenced by row and column addresses within the array. Display of data in cellular form is very easy because there are many display programs for displaying raster formatted data. Because of the popularity of raster formatted data in computer graphics and data communication, an abundance of software has been developed to process cellular data. Implementation of GIS using cellular data structure is less expensive because many software packages which can be used already exist.

Another advantage of the cellular data structure is its compatibility with remote sensing technology. Satellite imagery often is used in GIS for updating cartographic data and analysis of environmental changes. Satellite imageries such as SPOT and LANDSAT imageries are in raster format and can be integrated easily with other cellular data. The display program for cellular maps can easily be adapted to display raster satellite imageries.

Optical scanners used to scan hardcopy maps for inputting cartographic data into a computer digital format produce geographic data in cellular form [37, 8]. This could potentially lead to increased availability of geographic data in cellular format.

2.1.1.1 Thematic Overlays

As we said, each cell in a cellular data structure represents a square area in the real world. Associated with that square area of the world are different geographic attributes such as soil type, elevation, slope, etc. These different attributes are represented using the concept of thematic overlays. Because each cell within an array can take on only one value, each geographic attribute must be represented by a different array or overlay. In this construct, soil type would be described by a soil type overlay that is an array of cells with cell values denoting the soil type. e.g., a cell value of 1 for gravel, a cell value of 2 for clay, etc. Geographic data in cellular data structure therefore are represented in thematic overlays, with each overlay having a geographic "theme." A picture illustrating this concept is shown in figure 1 where different thematic overlays

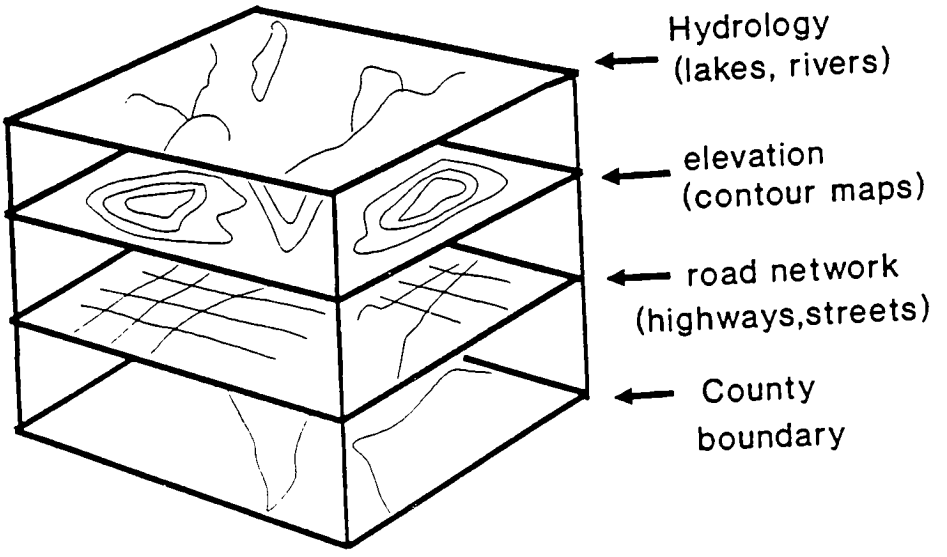


Figure 1: Thematic Overlays

for the same area are stacked one upon another.

In actual implementation, a cellular system could store many different thematic overlays into one physical array, i.e., an array of records. If the value range of a geographic attribute is small, we can compress different thematic overlays into a range of bits. For example, if each cell contains 1 byte then bits 1-3 could be used to store soil type information, bits 4-5 could be used to store slope information, etc.

2.1.1.2 Disadvantages of Cellular Data Structure

GIS using cellular data structure are prevalent because of the advantages

described earlier, ease of implementation, compatibility with satellite imageries, and structure simplicity. The structure does, however, have these disadvantages:

- Lacks flexibility - the resolution is fixed.
- Difficulties due to quantizing.
- Inefficient space utilization.
- Boolean operations on thematic overlays can be time consuming.
- Data input and output quality are often inferior.
- Unsuitability for range queries.

Once a datum is collected and stored in cellular form in a certain resolution then the resolution for that data is fixed, i.e., a higher resolution version of that datum cannot be obtained without going through the process of recapturing the datum in higher resolution. For example United State Geological Survey (USGS) currently have elevation data for the entire United States in cellular form at the resolution of 1:50,000 map scale. If elevation data of 1:24,000 map scale were to be needed for analysis purposes, then the area would have to be resurveyed to yield the higher resolution data.

As mentioned earlier, data in cellular format is quantized and not continuous. This can lead to problems in estimation of distances and areas [8]. This is illustrated in figure 2, where the true distance between a and c is 5, but in a cellular format this distance could be 4 or 7.

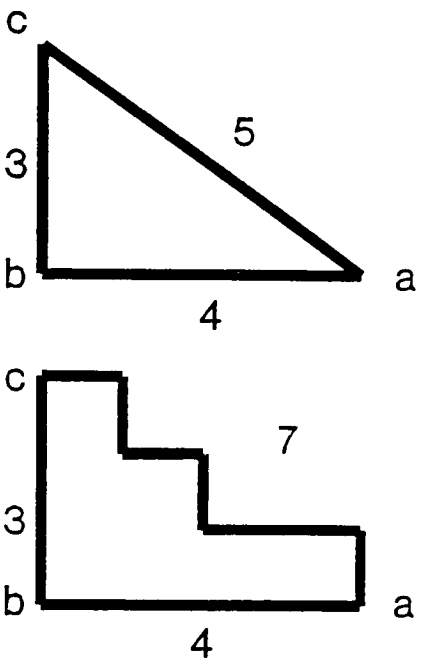


Figure 2: Quantizing Error

The major disadvantage of cellular data structure is the space requirement for storing the thematic overlays. For an overlay of n rows and m columns the space requirement is n $\times$ m $\times$ (bytes/cell). A 1024 by 1024 overlay of 1 byte size cell would therefore require $\geq$ 1 megabytes of storage. This can be expensive if the geographic database contains many thematic layers.

Boolean operations on geographic data are fundamental to many of the spatial queries used in GIS. In cellular data structure, a boolean operation consists of applying the logical AND, OR, NOT to thematic overlays. An example of a boolean operation query would be, "Find all the areas that have soil type clay and slope $\leq$ 5 degrees." This would translate to an AND of the soil type overlay and slope overlay. To perform a boolean operation such as AND on 2 thematic overlays, corresponding cells of each layer are examined to determine the cell value of the resulting overlay. This is a time intensive operation since all of the cells of each overlay must be examined. For an overlay of 1024 by 1024 cells this would require $\gg$ 1 million fetch and compare operations for each layer. For most cellular data structure systems, boolean operation is the primary bottleneck in analysis.

Range searching is another common operation on geographic data. A range search query is a search for all objects within a specified area. An example might be, "Find all wheat growing areas within 10 miles of a particular location." To perform range searching on a thematic overlay would also require examining each cell within the query range to obtain the result. This is undesirable as the search range could be large, leading to wasteful processing of non relevant areas.

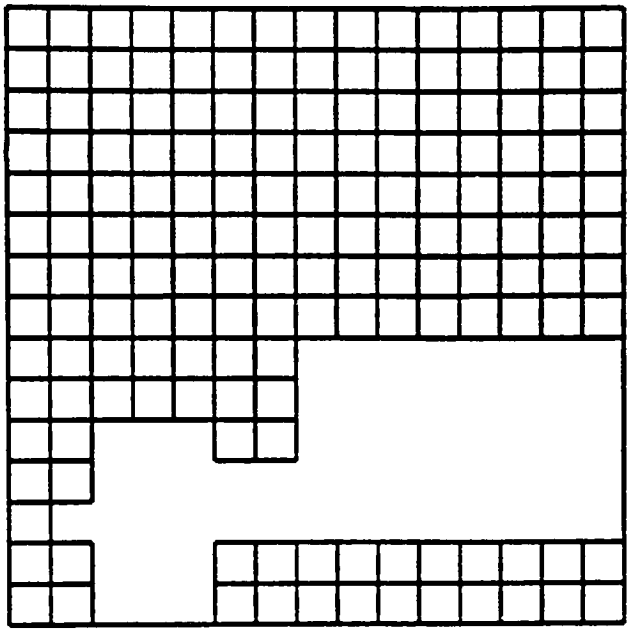


Figure 3: A Simple Region

2.1.1.3 Data Compression Techniques for Cellular Data Structure

Some of the disadvantages described earlier could be alleviated by using specialized techniques. We present two well known techniques, run length encoding and block codes, that are employed to minimize the space requirement of storing thematic overlays. One of the techniques, run-length encoding can also speed up the time required to perform boolean operations on map layers.

Run-Length Encoding - Run-length encoding [8] exploits the horizontal spatial coherence of the thematic overlay to obtain a more concise representation. The algorithm is simple and can be easily implemented. A row of the thematic

overlay is scanned from left to right recording blocks of homogeneous cells as a series of codes. Thus a thematic overlay can be captured in a succinct way by a series of codes representing each row of the array. For example using the run-length encoding method, the thematic overlay shown in figure 3 is represented by the following series of codes:

row 9 8,15
row 10 8,15
row 11 3,5 8,15
row 12 3,15
row 13 2,15
row 14 3,5
row 15 3,5

In this example the space needed to store 225 individual cell values for the thematic overlay can be reduced to the space needed to store 16 numbers

Given sufficiently homogeneous data, the run-length encoding method can produce enormous savings in the amount of space required to store thematic overlays. Boolean operations on thematic overlays can also be more efficiently performed because instead of comparing individual cells it is necessary only to compare a succinct series of codes for the overlay, thereby minimizing the time needed to perform boolean operations.

Because of these advantages and the simplicity of the algorithm, run length encoding is widely used by existing cellular data structure systems. It should be

noted, however, that run length encoding doesn't always produce savings. If the data are very non-homogeneous, as in a "checkerboard" pattern, then run length encoding may require more space due to the overhead of storing the codes.

Block Codes - Block codes [8] extend the run-length encoding idea to two dimensions by exploiting the spatial coherence of the data. The relevant regions within a thematic overlay are decomposed into square blocks described by their origin and sides. These blocks constitute the block codes representation for the thematic overlay.

As in the run-length encoding method, the success of this technique is very dependent upon the degree of homogeneity of the data. The more homogeneous the data, the higher will be the compression ratio.

Because the block codes method of data compression requires more processing time on the overlay to determine its equivalent block codes it appears to be used less often than run-length encoding for data compression.

2.1.2 Vector Data Structure

Vector or polygonal representation of geographic data uses vertices as its basic unit of storage. A vertex is a coordinate pair corresponding to a coordinate system. All three general types of geographical data (point, lines, areas) can be represented using vertices. Point data can be described by a single vertex. Linear data such as roads or rivers can be described by an ordered list of vertices. Area data, such as forest boundaries can be represented by a list of vertices describing

the boundaries.

High quality cartographic data are usually input into digital form using vector format. An instrument called the "digitizer" is used to trace the map data. The operator places a cursor to trace the data using clicks on the cursor to enter the vertices. Therefore, a map is digitized by inputting a set of vertices describing the spatial properties of the data.

Because the vector data structure is the one that is closest to the data input source, many standard cartographic data are in vector form. An example is the format of the United States Geological Survey (USGS) Digital Line Graph (DLG) data, in which geographic data are stored in a vector data structure.

The output of systems using vector data structure usually is superior to that produced by cellular systems because the pen plotter can produce drawings without the staircase effects characteristic of some raster printers. However, as a result of advances in raster printing technology, some of the higher priced raster printers can now produce quality that approaches that of pen plotters.

Range searching on vector data can be accomplished by clipping the line segments described by the vertices against the search range area. This can be done quickly if the vertices list is ordered by spatial location.

2.1.2.1 Disadvantages of Vector Data Structure

Although the vector data structure is widely used and is standard for many cartographic databases, it does have some drawbacks.

- Statistical analysis can be complicated.
- Boolean operations on polygons is time consuming.
- Vector systems are more difficult and expensive to implement.

Statistical and quantitative analysis using a vector data structure is difficult because of the complexity of algorithms for analysis of irregular geometric shapes. For example, to calculate the area of an irregularly shaped region described by its vertices is non-trivial, whereas this operation can be done easily in cellular format by simple counting of cells. Similarly, quantitative analysis such as calculation of perimeter, sums and average are also much more difficult in vector format.

Boolean operations on geographic data in vector format is also difficult. Despite advances in algorithm development for performing polygon overlays [56, 61, 62, 63] the calculation of the union and intersection of irregularly shaped polygons still is highly time consuming. This is in contrast to the simple algorithm used in boolean operations on thematic cellular overlays, where only a simple comparison of cells is needed.

Because analysis of data in a vector data structure usually require implementation of complicated algorithms vector systems generally are much more expensive to implement than their cellular data structure system counter-parts.

2.2 Hierarchical Data Structures

In this section we describe several hierarchical data structures that have been proposed for geographic data processing. These structures are based on the principle of recursive decomposition of the spatial domain. Hierarchical data structures are useful because they enable fast pruning of non-relevant areas from consideration. This is accomplished by variable resolution at each level; e.g., the topmost level has the coarsest resolution, with the resolution becoming more detailed as the lower levels are approached. This makes possible rapid determination of areas of potential interest, which can be explored in more detail by descending into lower levels of the data structure.

2.2.1 Quadtrees

Quadtrees have been extensively studied for application to geographic information processing [54, 51, 52, 46, 49, 45, 50]. A quadtree is a hierarchical data structure based on the principle of recursive decomposition of two dimensional space.

The term "quadtree" was first used by Finkel and Bentley in [18] for a data structure used in representation of points data. Hunter and Steiglitz [28] describe the quadtree structure, also known as a "region quadtree," as one in which polygonal regions can be represented in a quadtree. Most references to quadtrees are to region quadtrees, which are based on the principle of *regular* recursive decomposition in which two dimensional space is recursively partitioned into

equal sized areas.

To construct a region quadtree from a binary array representing a region, the entire array is first examined to determine if it is homogeneous (i.e., consisting entirely of equivalent pixels). If the array is not homogeneous, it is divided into equal sized quadrants. Each quadrant is recursively examined to determine if it is homogeneous. Processing stops for homogeneous quadrants. Non-homogeneous quadrants are further subdivided, and the process continues until only homogeneous blocks are obtained, possibly down to the individual pixel. Thus a quadtree is a tree representation of two dimensional space.

In figure 4 an example cellular array describing a polygon region is shown: its corresponding quadtree is given in figure 5. In the region quadtree shown in figure 5 the root of the tree corresponds to the entire array. Because the entire array is non homogeneous, it is decomposed into equal-sized quadrants of NW, NE, SW, SE, representing the northwest, northeast, southwest, and southeast quadrants of the binary array, respectively. At this point it can be seen that the NW quadrant is homogeneous, so decomposition of the NW quadrant is terminated and it is represented by a WHITE leaf node. The other quadrants are non-homogeneous and correspond to the gray nodes C, D, and E. Decomposition continues until only homogeneous blocks are obtained. All non-leaf nodes in a region quadtree are GRAY nodes. The leaf nodes in a quadtree are either BLACK or WHITE depending on whether the quadrant consists of only 1's or 0's.

It can be seen that a region quadtree can be a space efficient method of repre-

senting region data if the region being represented is sufficiently homogeneous. One of the reasons for interest in quadtrees during earlier studies has been its ability to exploit the spatial coherence of data to enable space savings.

2.2.1.1 Quadtree for Spatial Analysis

Another important reason for the interest in quadtrees is the ease of performing boolean operations on data in quadtree representation [28, 57]. To perform a boolean operation such as AND of two regions represented by quadtrees requires only a parallel traversal of the quadtrees to examine the corresponding nodes and construction of the resulting quadtree. This can be much more efficient than comparing individual cells of the regions, because only the quadrants of the regions represented by quadtree nodes need to be compare.

Algorithms also exist to allow for many of the other types of spatial analysis common in GIS applications using quadtrees. These include connected component labeling, perimeter calculation, medial axis transform, and others [46].

2.2.1.2 Shift Variant Property of Quadtrees

Quadtrees are shift variant; i.e., shifting a region to a different location within an image will generate a very different quadtree. Figure 6 shows two otherwise identical regions differing only by a translation; below them are their respective quadtree representations. As can be seen, a very different quadtree can result from a simple shifting of the region.

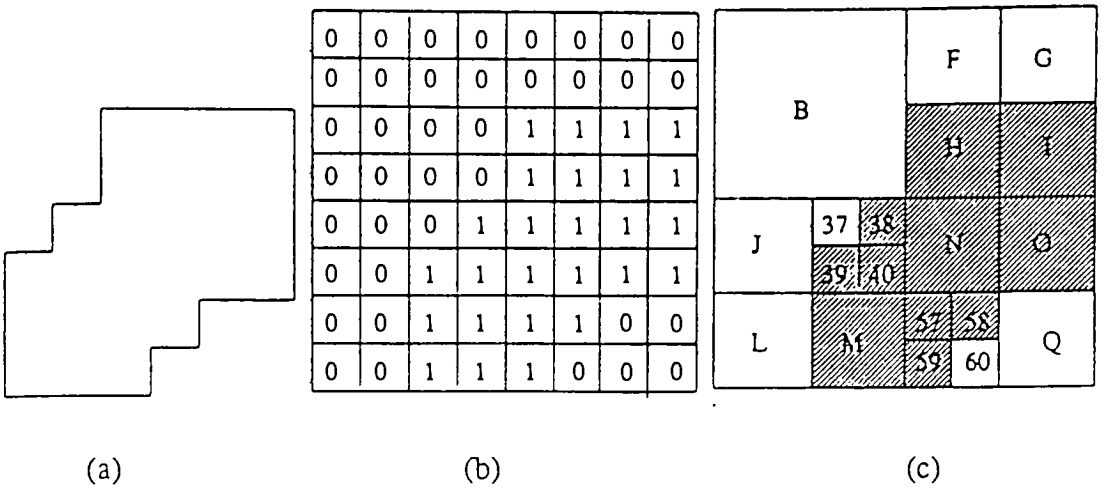


Figure 4: Binary Array

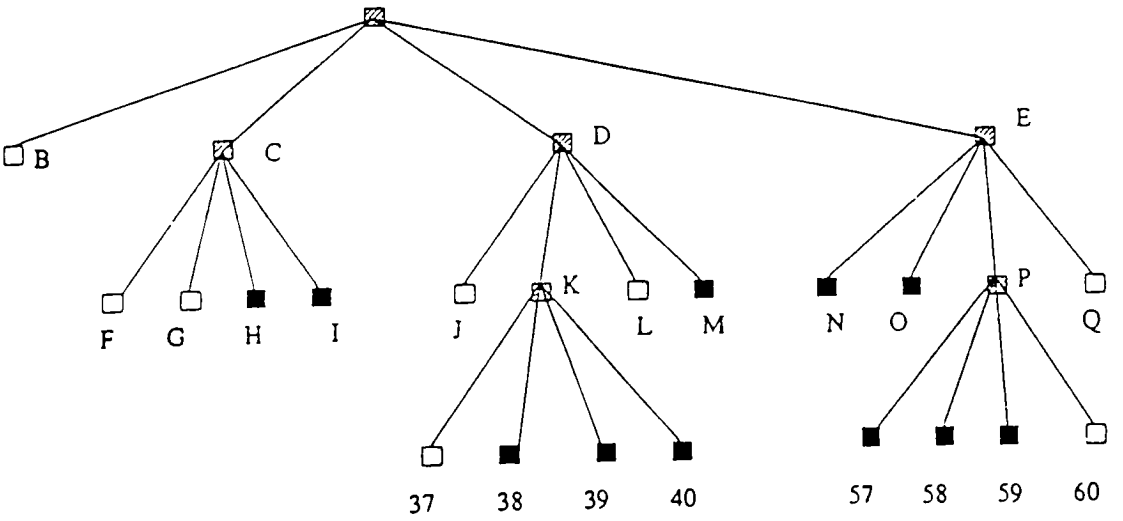


Figure 5: Corresponding Quadtree

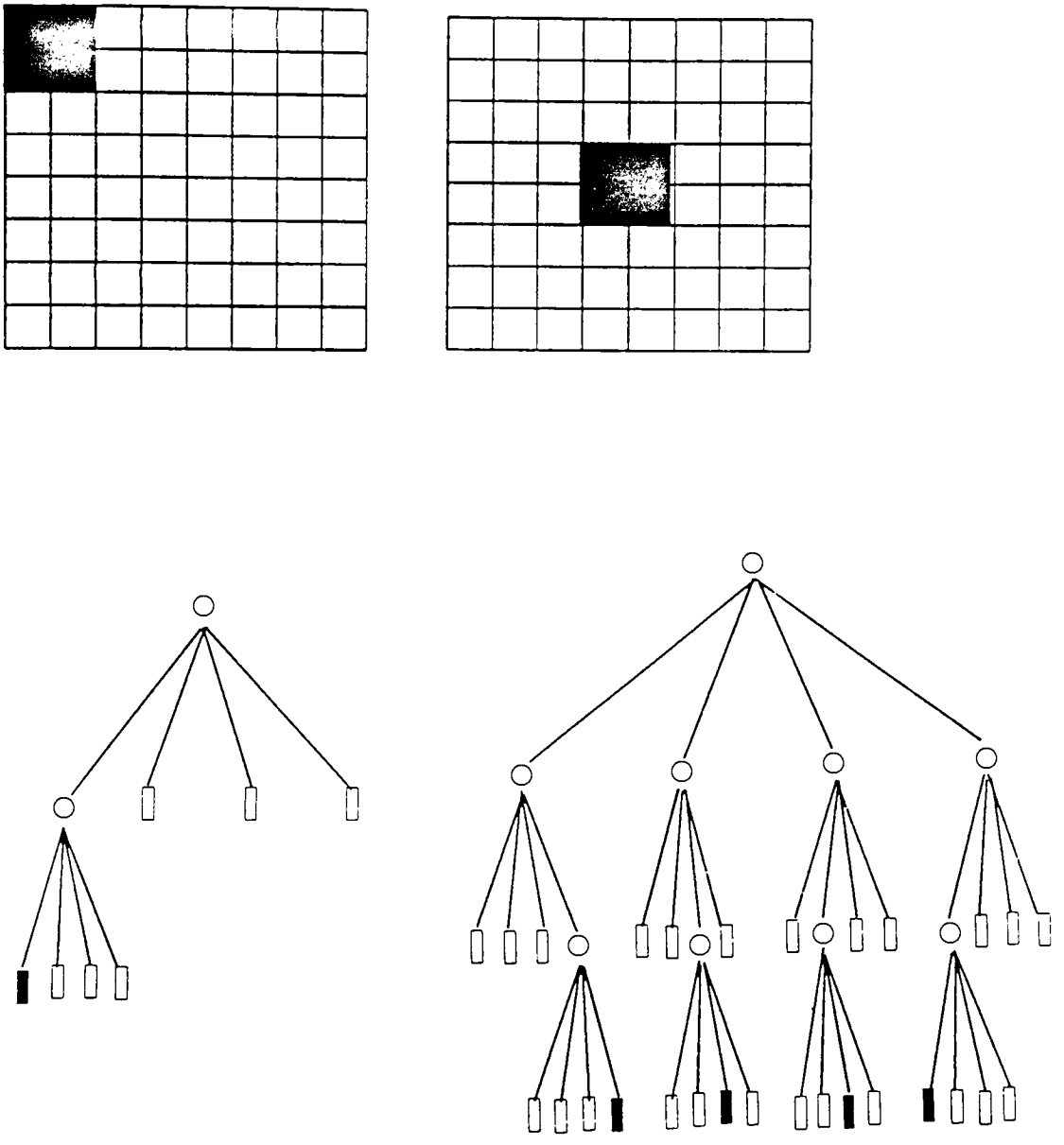


Figure 6: Shift Variancy

Data structures that are shift variant are more difficult to manage since any minor translation of data could require recomputation and reorganization. This is significant because geographic data no longer are static for long periods of time. Development of remote sensing technology has enabled dynamic changes in our environment to be rapidly integrated into the geographic database. This requires frequent updating and minor adjustments in the geographic database to reflect environmental changes. The shift variance property also has consequences for the space efficiency of quadtrees. Dyer [16] has shown that, depending on where the region is located within the array being represented, a 2^n by 2^n region could be represented by one node of the quadtree or $O(2^n)$ nodes.

2.2.1.3 Current Status of Quadtrees

A major impetus for the development and interest in quadtrees has been their perceived ability to provide space efficiency in the storing of geographic data. Quadtrees have been viewed as a data structure that can cut down on the enormous space requirements of geographic data storage. Preliminary results using quadtrees for data compression of block letters and other types of graphic data have produced encouraging results [28]. This and other researchers' results spurred the development of a quadtree-based GIS in [50]. Quadtrees have been shown to be effective for storing simple geographic data such as road network maps, city boundaries and other cartographic data.

As research and applications of quadtrees have increased, however, some disadvantages have become apparent. Early quadtree descriptions [28, 46] used

pointer structures, in which nodes store the pointer addresses of their children. The pointer structure uses explicit links between a node and its sons, thereby facilitating access to its leaves, but this is at the cost of the space used to store the pointer addresses. Geographic data are characterized by very irregularly shaped geometric objects and are often highly non-homogeneous. In many cases the internal storage space needed for the pointers of the quadtree has exceeded that realized from the quadtree via data compression. This has led to development of many alternative quadtree representations such as the linear quadtree of Gargantini[21, 22], the tree codes of Oliver and Wiseman [39], and the DF expression of Kawaguchi and Endo [31]. All of these alternative quadtree structures have eliminated the explicit pointers of the original pointer-based structure and replaced them with an implicit tree structure by using codes, as in the linear quadtree, and string expressions as in the DF expression.

Geographic data usually require significant amounts of space for storage that must be on secondary storage devices. This means data structures used to manage geographic data must make efficient use of secondary storage devices by minimizing disk access. Because quadtrees are not height balanced, they cannot guarantee efficiency in disk accessing. This is a major drawback in the application of quadtrees for geographic information processing.

Boolean operations on aligned quadtrees (i.e., quadtrees with identical origins and size) use a simple algorithm that traverse the quadtrees. On unaligned quadtrees (i.e., quadtrees which have different origin and are of different sizes) is considerably more difficult. Windowing a subsection of a quadtree similarly

requires a more complicated algorithm than that used for cellular and vector data structures. In cellular format, windowing merely requires taking a subsection of a file. In vector format, windowing only requires clipping lines against a window. On a quadtree, windowing not only requires a search of the tree but also merging of the results [47].

2.2.2 Alternative Quadtree Representation

The original presentation of quadtrees were called pointer-based quadtrees and used explicit pointers to link a node to its descendants[28]. Using pointers to implement a tree data structure is very common. However, for a binary array that is very non homogeneous and exhibits a "checkerboard" pattern the memory expended for internal storage of pointers can exceed the binary array representation.

We present alternative quadtree representations, developed by various researchers, that eliminate the explicit pointers and derive more space efficient alternative quadtree representations.

2.2.2.1 Linear Quadtrees

The linear quadtree of Gargantini [21, 22] is a pointerless quadtree representation. It uses "quaternary" codes to encode the only the locations of black nodes. In this scheme each quadrant of a 2^n by 2^n cellular array is represented by a n digit code. For example, an array of dimension 2^3 by 2^3 would require

3 digits to encode a quadrant.

By elimination of the pointers, linear quadtrees have been shown to reduce the storage requirements of quadtrees by 66 percent to 90 percent. Boolean operations running time on linear quadtrees is comparable with that using pointer-based quadtrees [22]. Rotation by 90-degree increments and neighbor-finding procedures are actually more efficient using linear quadtrees.

2.2.2.2 DF Expressions

DF (Depth-First) expressions of Kawaguchi and Endo [31] were developed mainly as a data compression technique for binary images. They have the structure of a string made up of "(", "B", and "W" characters, representing gray, black, and white nodes respectively. An equivalent DF expression for a quadtree can be obtained by making a pre-order traversal of the quadtree and writing down the sequence of nodes visited. Because there are only three possible values for each element in the string only two bits are needed to encode one node of the quadtree. Thus DF expressions can be very space efficient. Empirical results in [31] have shown that they offer better compression factors than the following facsimile techniques.

- Block Coding.
- Two Dimensional Prediction.
- Run-Length Encoding.

In addition, algorithm are available so that DF expressions permit boolean operations and other spatial analysis.

2.2.3 Strip Trees

The strip trees [4] of Ballard is a hierarchical data structure that uses enclosing rectangles to store a curve. In [4] strip trees are shown to be an effective data structure for geographic data because they can handle curves and areas. A strip tree is a binary tree with the root representing the bounding rectangle that encompasses the entire curve. It is hierarchical because the decomposition of the curve produces successively finer resolution of the curve segments. Each node of a strip tree describes a "strip" or rectangle which contains a segment of the curve. Each node's rectangular region always encloses the curve segments stored in its children. Thus the root of a strip tree always represents the entire curve, with successive lower levels of the tree representing progressively smaller segments of the curve.

Ballard [4] presents two algorithms for generating a strip tree for a curve. One algorithm called "AO digitization" constructs the tree in a top down manner by first finding the strip for the root of the tree and then recursively constructing lower levels of the tree until the level of resolution w^* is reached.

Resolution level w^* describes the minimum width for the strip (or rectangle) such that all leaf nodes of a strip tree that are at resolution w^* will have strips whose width $\leq w^*$.

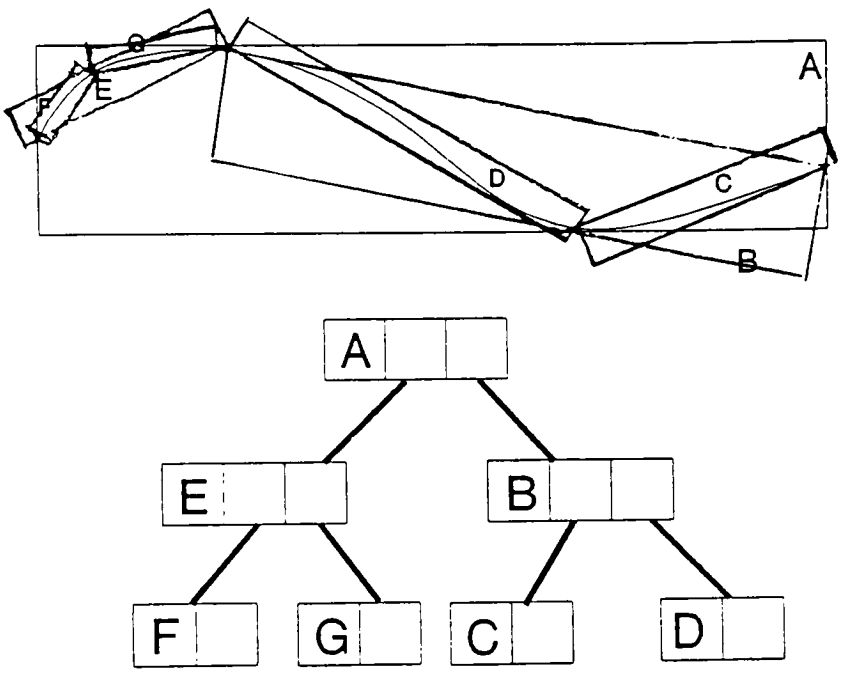


Figure 7: Strip Tree

The second algorithm for strip tree construction, called "AO/ digitization," builds the tree from the bottom up by first obtaining the strips for the leaf level satisfying the condition that the width of the leaf strip $\leq w^*$ and then pairing the leaf nodes with successively larger strips until the root strip is generated.

Figure 7 shows a curve and the hierarchically ordered strips used to cover it. The strip tree corresponding to the curve is shown below it.

In [4] Ballard has shown that a strip tree can facilitate many types of analysis

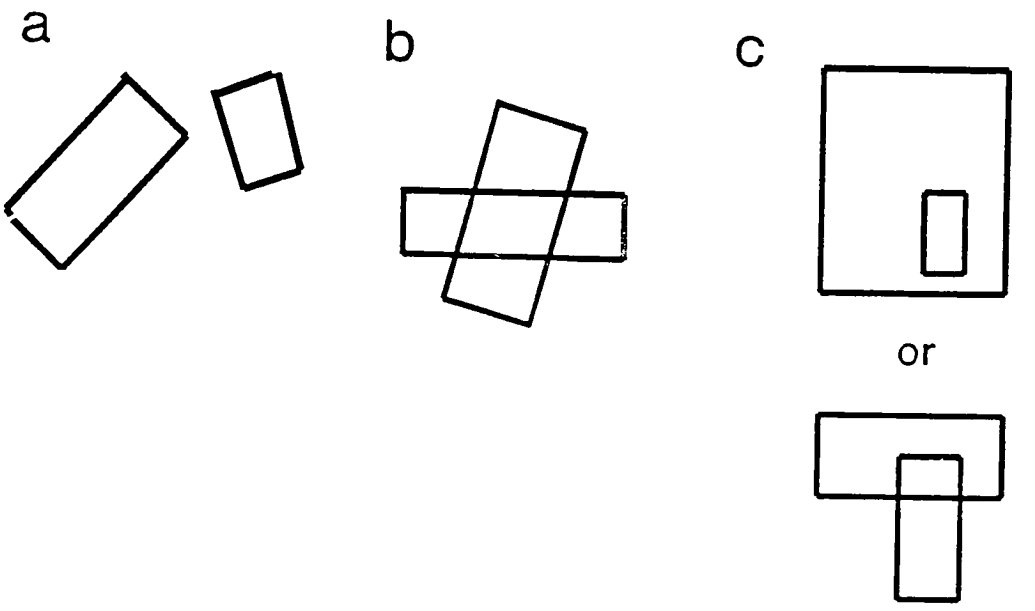


Figure 8: Three Scenarios For Strip Tree Intersection

common to geographic data processing. Boolean operations on both curves and areas can be efficiently processed using strip trees. For example, suppose it is necessary to determine whether two roads represented by their respective strip trees intersect? This translates to an intersection of their strip trees. To intersect two strip trees only three possible situations can occur and these are shown in figure 8.

The relationship between two strip trees can be characterized by any of cases A, B, or C shown in figure 8. If the strip trees have the relationship shown in case A then they don't intersect. If the strip trees have the relationship shown

in case B then they must intersect. It is only in case C when further searching is necessary to determine their intersection. Thus the strip tree can rapidly detect cases where no intersection is possible, and the opposite condition, which is when intersection must occur.

Strip trees can also facilitate calculation of the length of a curve, the proximity of a point to a curve, or can determine a point in a polygon query.

Two main disadvantages of the strip trees are the large amount of space overhead needed for creating the strip tree structure and that a closed curve requires special handling.

2.2.4 BSPR

Binary Searchable Polygonal Representation (BSPR) [9] is a hierarchical data structure that approximates curves and area boundaries by a set of upright rectangles. BSPR is similar to a strip tree except that its rectangles can only have one orientation (upright), whereas the strip tree's rectangles can have any orientation.

BSPR is also a binary tree with the root representing the entire curve, and lower levels that represents subsections of the curve. Burton describes an algorithm for constructing BSPR that uses bottom-up processing of the curve. The curve is first broken down into simple sections each of which represents a segment of the curve that is monotonic in both the x and the y axis. These simple sections are paired to construct compound sections corresponding to a higher level of the tree, and the pairing continues until the compound section for the

root which encompass the entire curve is created. Burton shows how to process a point in polygon queries and curve intersection queries using BSPR.

2.3 Other Structures

This section covers data structures that do not fit under the headings described earlier. Two data structures, k-d trees and grid file, which had an important effect on the development of data structures for geographic data processing are presented. k-d trees have been extensively researched and found to be very efficient for answering range queries. Many empirical studies have been conducted comparing their performance with that of quadtrees [34, 42]. The k-d tree is a well known structure that is still generating interest for applications in geographic information processing.

The grid file of Nivergelt et al. [38] is a significant structure for geographic data representation in that its developers recognized the importance of using data structure to minimize disk access in answering queries. Its structure guarantees that at most two disk accesses are required for exact match queries (point queries).

2.3.1 k-d Trees

k-d tree [6], a data structure developed by Bentley for storage and retrieval of multidimensional data, is a multidimensional binary search tree which performs

associative searching (i.e., retrieval by secondary keys). k-d tree have been extensively studied in the literature [42, 59]. In [59] they were shown to be useful for representing geographic data. In [42] empirical results showed that k-d trees are superior to quadtrees for region searching.

The k-d tree is a binary tree in which each record is stored as a node in the tree. It differs from a traditional one-dimensional binary search tree in that different levels of the k-d tree use different keys for ordering the indices. The k in k-d tree denotes the dimensionality, which in turn defines the number of keys. For example, a 2-d tree can be used to store two dimensional data in cartesian space and would have 2 keys, x and y. Bentley called keys, *discriminators*, and each level of the tree is associated with a *discriminator*. Each discriminator is assigned a unique number from 0 to k-1. The assignment is arbitrary but once assigned it must not be changed. For example, in a 2-d tree x could be assigned as discriminator 0 and y as discriminator 1. The root of the tree is associated with the discriminator 0 and its sons' level is associated with discriminator 1, and so on until the kth level of the tree is reached. The kth level is associated with discriminator k-1 but the k+1 level is again associated with discriminator 0. Essentially, a different discriminator is used at each successive level of the tree following in order from 0 to k-1. After every k-1 discriminator has been used the cycle repeats.

Figure 9 illustrates a set of data points and their organization in a 2-d tree. A corresponding 2-d tree representing those data points is shown in figure 10. Each data point partitions two dimensional space in either the x or the y direction,

depending on the discriminator used. The k-d tree in figure 10 demonstrates the cycling of discriminators mentioned earlier. The root node (level 1) of the tree is partitioned using the discriminator 0 (x) and its sons (level 2) is partitioned using the discriminator 1 (y) and so on.

The insertion algorithm for the k-d tree is straightforward and is similar to that for one dimensional binary search trees. The algorithm basically determines the discriminator used at each level, makes the comparison and continues the traversal until it reaches the bottom of the tree, where the record is inserted. Two strategies can be used for keeping track of the discriminator. One strategy is to store the discriminator with the node as in the original Bentley paper [6]. This makes the discriminator information explicit but wastes space. A more space efficient strategy is to note that the discriminator used for a particular level of the tree is fixed. Therefore the discriminator for any level can be found by (level mod k).

k-d trees do not partition space in a regular manner. The order in which the data records are inserted into a k-d tree can affect the resulting shape of the k-d tree. Bentley [6] has shown that if the points are inserted in random order into a k-d tree then the total path length (TPL) for that tree would be O(N log N) where N= number of data points. But if the points are inserted in order (ordered by spatial location, e.g., increasing in x or y) then the TPL will be large, leading to degradation in performance for the k-d tree.

To understand how ordering the data points can lead to growth of the TPL we make the following characterizations. If we insert records x1, x2 ... xk which are

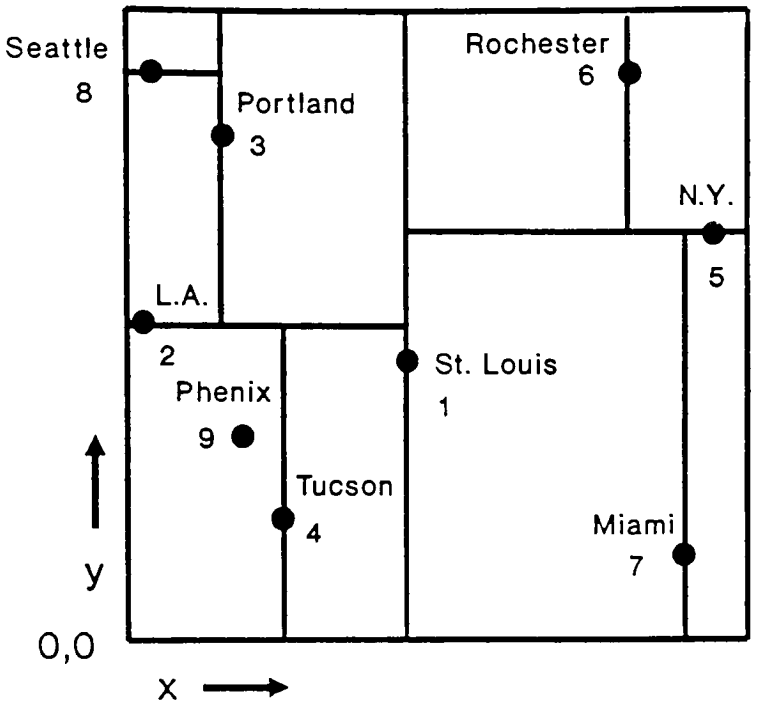
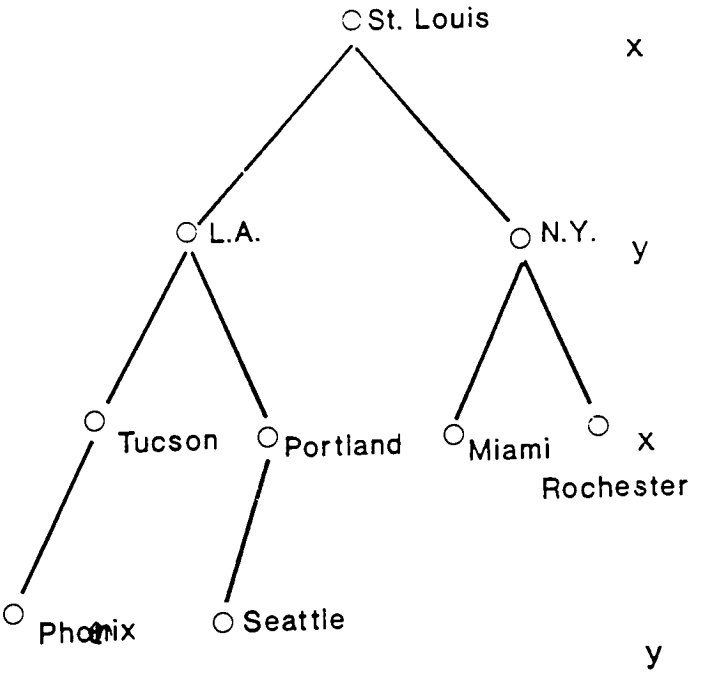


Figure 9: Input Data To Create A k-d Tree



increasing in their x value, then the resulting k-d tree will be a comb. A comb is a tree that is not height balanced, and if any node within a comb has more than one subtree as descendants then only one of them can have more than one son [5]. The worst case for a 2-d tree is if the ordering is in both dimensions. An illustration of this in the 2-d tree case would be if the data points were to lie in a diagonal line and if the insertion were to be made in order from one diagonal to the other end. This is illustrated by figure 11 showing the data points and the k-d tree (shown in figure 10) built by inserting records which are ordered both in the x dimension and the y dimension. In this particular case the x values are ordered in increasing value and the y values are ordered in decreasing x values. Thus when a k-d tree is built by inserting records that are in order, a unbalanced k-d tree with high TPL will be the result.

To prevent this type of worst case behavior and to minimize the TPL Bentley presented an algorithm to generate an "optimal k-d tree" in which the tree would be relatively balanced because the number of nodes on the left side of the tree differ by no more than one node from the number of nodes on the right side of the tree. The "optimal k-d tree" is a static structure since all records must be known a priori. The algorithm is a recursive procedure which repeatedly selects a median value record that will bisect the data records. This algorithm has a running time of O(N log N) and would minimize the TPL. There are other static structures which also attempt to balance the k-d tree. One example of this is the adaptive k-d tree of Friedman [20].

The search performance of the k-d tree makes it a very useful data structure

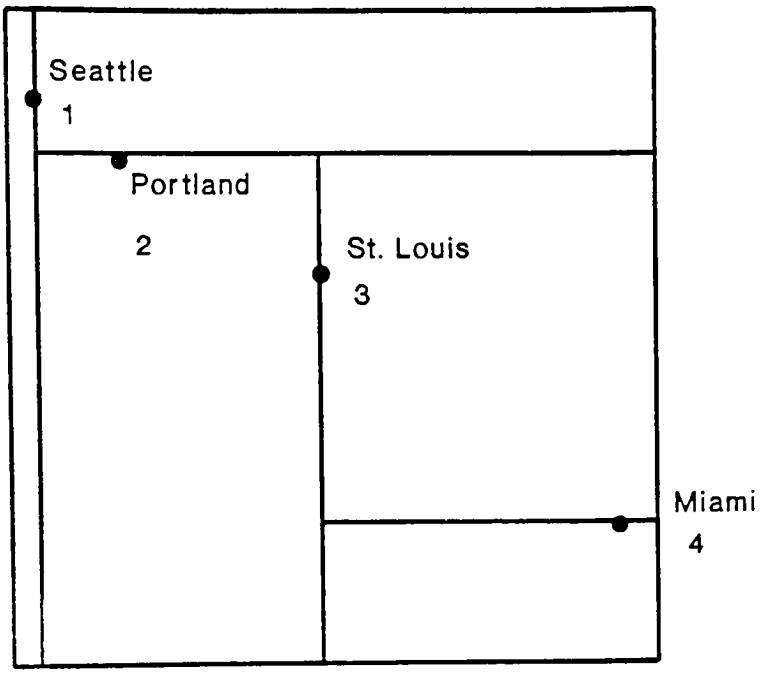


Figure 11: Set of Ordered Data Points

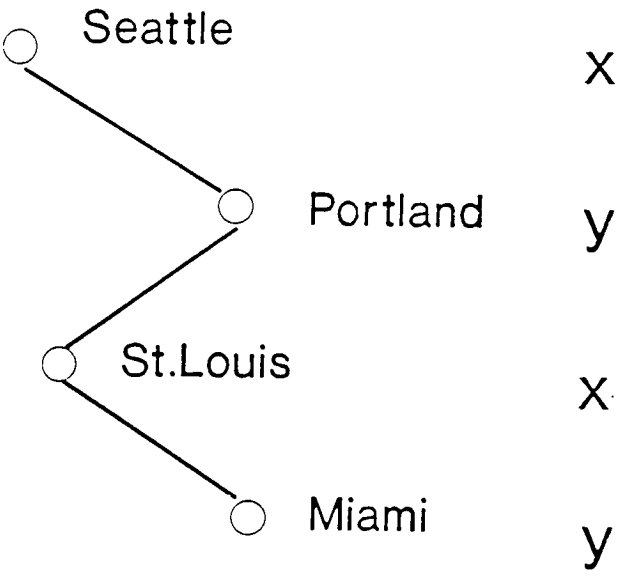


Figure 12: Worst Case k-d Tree From Ordered Data

in geographic applications. In Lee and Wong [34] the k-d tree is shown to have range search cost of $O(k \times N^{1-1/k})$, where N = number of records in the k-d tree. Empirical results in [42] have also shown k-d tree to be superior to quadtree in region searches. The successful application of k-d tree for storage and retrieval of geographic data has been demonstrated in [59]. Thus k-d tree is a data structure that offers significant promise as a data structure for GIS applications.

The main disadvantages of the k-d tree are the time required to delete a node in a k-d tree and the lack of a dynamic structure that is height balanced. It is much more difficult to delete a node in a k-d tree than in a one dimensional search tree. Bentley has shown that the cost of deleting a randomly selected node from a k-d tree is O(log N), but the cost will be significantly greater if the node being deleted is a root node, since a search must be conducted to find a suitable replacement. The lack of a dynamic k-d tree that is height balanced is a drawback since geographic data frequently must be stored on disk. This requires a data structure that can minimize disk accesses.

2.3.2 Grid File

A data structure derived from the bitmap approach of organizing data is the grid file of Nivergelt, Hinterburger, and Sevcik [38]. The grid file is similar to the grid or cellular method in which cells are created by partitioning the space domain except that in a grid file the cells are not of uniform size.

The grid file was developed with two main principles in mind. The two disk access principle states that an exact match query in which a single record is retrieved should require at most two disk accesses. The second principle is the efficient range search query principle, which also tries to minimize disk access during range queries by organizing data records so those that are spatially near are in the same storage block. Grid file structure does guarantee that single-point queries require at most two disk accesses, but range queries still require considerably more disk accesses.

The space being represented by a grid file is partitioned into a grid directory containing grid blocks (whose counterparts are cells in cellular structures) of non uniform sizes. The grid directory is composed of two parts. One component of the grid directory consists of a set of k one-dimensional arrays termed linear scales. The linear scales partition the space dimensions which they represent, which in turn defines the resolution of the grid blocks in the grid directory.

Therefore each array of the linear scale defines the partition for one dimension. For example, for a grid file storing two dimensional data there would be two arrays representing partitions in the x and y directions. The linear scales are assumed to be small enough to be stored in main memory.

The other component of the grid directory is a k-dimensional array with the number of elements defined by the linear scales. For example, for two-dimensional data (k=2) the number of elements in the k-dimensional array will be $n_x \times n_y$ where n_x and n_y are the number of partitions in the x and y linear scale respectively. Each element within this k-dimensional array is a pointer to

the data bucket storing the actual data. Each bucket is assumed to have capacity to store 10 - 1000 records.

In figure 13 a collection of data points is shown and their corresponding grid file representation is shown in figure 14. As stated earlier, the grid directory contains two components, the linear keys represented by the two one-dimensional arrays and the two-dimensional array stores the grid blocks.

To demonstrate the search procedure for a grid file we give the following example. To search for the record at location (40, 60) we examine the linear keys and determine that the record is in grid block (2, 1). We access the grid block (2, 1), find the address of the physical disk block containing the record, and retrieve the record. Thus we can see that the two disk accesses principle for a single-point query is satisfied, since we can determine the grid block needed to be accessed from the linear scales stored in main memory. The assumption that the linear keys can be stored in main memory is justified because the linear key grows very slowly in relation to the number of records. The growth of the linear key is $O(N^{1/k})$ where N = number of records and k = dimensionality of data.

The grid file is a promising data structure for GIS because its developers recognized the characteristics of geographic data (mainly that their size usually precludes their being stored on secondary storage devices). But not enough empirical or theoretical evidence exists to show that the grid file can handle all forms of geographic data (such as curves and areas) efficiently, and algorithms for spatial analysis such as boolean operations, spatial proximity, etc, using grid

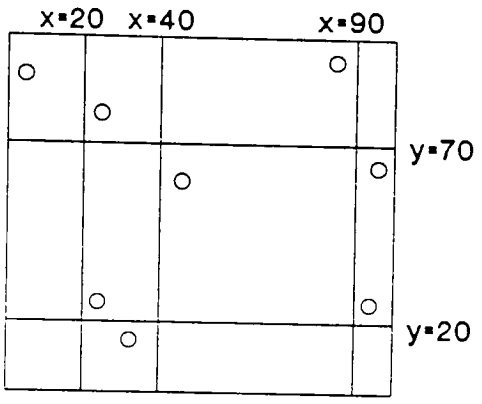


Figure 13: Data Points Organized in a Grid File

Linear keys

x	y
20	20
40	70
90	

Grid blocks

(1,2)	(2,2)	(3,2)	(4,2)
(1,1)	(2,1)	(3,1)	(4,1)

Figure 14: Grid File

files still are lacking.

2.4 Recent Developments

In this section we cover the relatively recent developments of spatial data structures for GIS. Recently, there has been great deal of interest in higher level data structures which can be used to manage geographic data efficiently. R-tree is a structure that can allow for the efficient storage and access of large amount of spatial data because of its B-tree like structure, which minimizes disk access. R-trees also have been used to implement extensions of relational databases into geographic applications. This is seen in the development of PSQL which is an SQL extension into spatial data processing described by Roussopoulos and Leifker [43]. We will described R-trees and their behavior and include a brief discussion on similar structures such as R^+ trees and cell trees.

2.4.1 R-trees

An R-tree [27, 43] is a hierarchically structured height-balanced tree similar to a B-tree. R-tree was developed for efficient storage and retrieval of multi-dimensional data. It is especially useful for range searching and is optimized for data which have large space requirements that must be on secondary storage devices. R-tree's efficiency for data stored on disk is attributed to its B-tree like structure, and B-tree performances on minimization of disk access can be guaranteed.

Instead of having an alphanumeric key as in a B-tree, each node of an R-tree has a rectangular region of coverage which denotes that only data items that are entirely covered by this rectangular region can possibly be located in its descendants. An R-tree is hierarchical in that rectangular regions in its upper nodes entirely cover all the rectangular regions of its corresponding children. This makes the search procedure for an R-tree simple and efficient.

To search for all data items within a query region, the search algorithm simply recursively descends the tree and compares the search region against the rectangular region of the current node, exploring only nodes that might contain relevant data items. The execution of this operation is efficient because the comparison is done only for rectangles, thus requiring only four comparisons. This speeds up the search process considerably, since large areas of non-interest can readily be rejected and clipping needs to be performed only on the necessary data items.

The leaf nodes in an R-tree [27], contain the index entries for the actual data items. Such an index entry has the form of:

(r, data-item-pointer)

where r is the minimum bounding rectangle (MBR) that entirely encloses the data-item and the data-item-pointer is the address or pointer to an actual data item stored in the R-tree. Non-leaf nodes contain high-level index entries (i.e., indices to indices) of the following form:

(R, (R-child1 .... R-childB) (child1....childB))

where R is the MBR that entirely encloses the node's children's rectangular

regions. R-child1 ... R-childB are the rectangular regions of the node's children and child1 ... childB are pointers to its children.

Like a B-tree, the order, b, of an R-tree is the minimum number of entries in a non-leaf node that is not the root (the minimum number of entries in the root is 2), and the maximum number, B, of entries allowed in a non-leaf node is equal to or larger than 2b (i.e., $b = \lfloor \frac{B}{2} \rfloor$). This makes it possible to define the fill criteria for the R-tree and the method used to keep the R-tree height balanced with random insertions. The R-tree, like the B-tree, has a fill criterion, i.e., each node of an R-tree must have between b and B entries unless it is the root. If the insertion of a data item will cause the total number of entries within a leaf node to exceed B, then the entries in the leaf node will have to be divided between this leaf and a new leaf node to fulfill the fill criterion. As in the B-tree, a node split will propagate up the tree and, if necessary, the root will be split, causing the height of the R-tree to increase by 1. Figure 16 depicts an example of an R-tree of order 2. This R-tree was constructed by inserting all the data items in figure 15 into the R-tree. (Data items are shown in lower case r, such as r1, r2 ...) The leaf nodes at the bottom of the R-tree point to the data items in secondary storage.

2.4.1.1 Performance Issues in R-trees

This subsection discusses some of the factors that affect R-tree performance, namely the selection algorithm for node splitting and the order of an R-tree.

As mentioned by Guttman [27] and Rousopolous and Leifker [43], the selec-

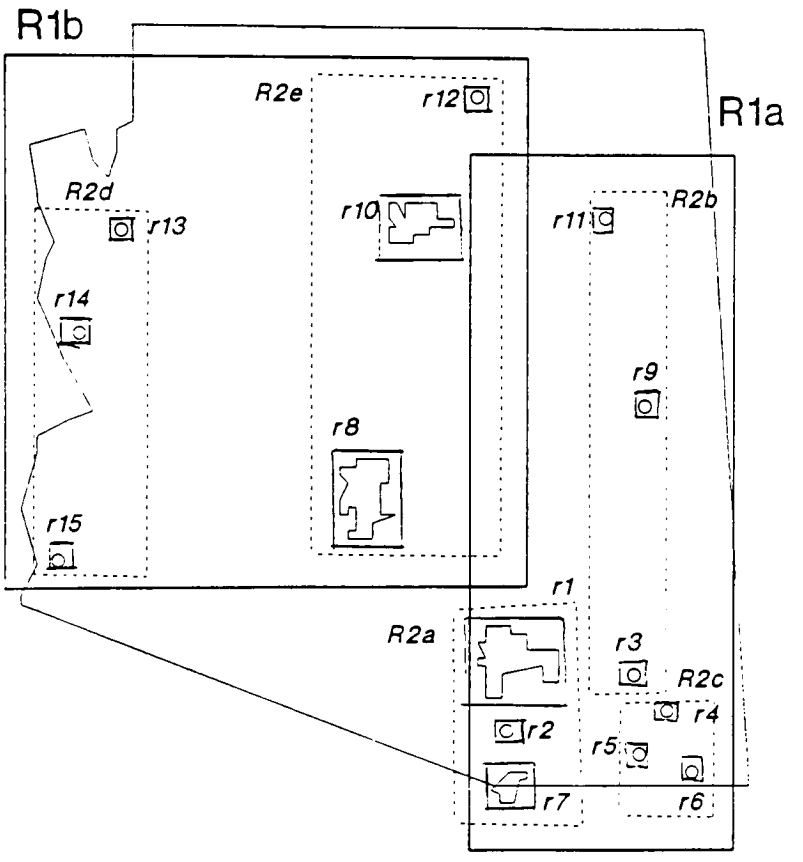


Figure 15: Some Arizona Cities as Data Points

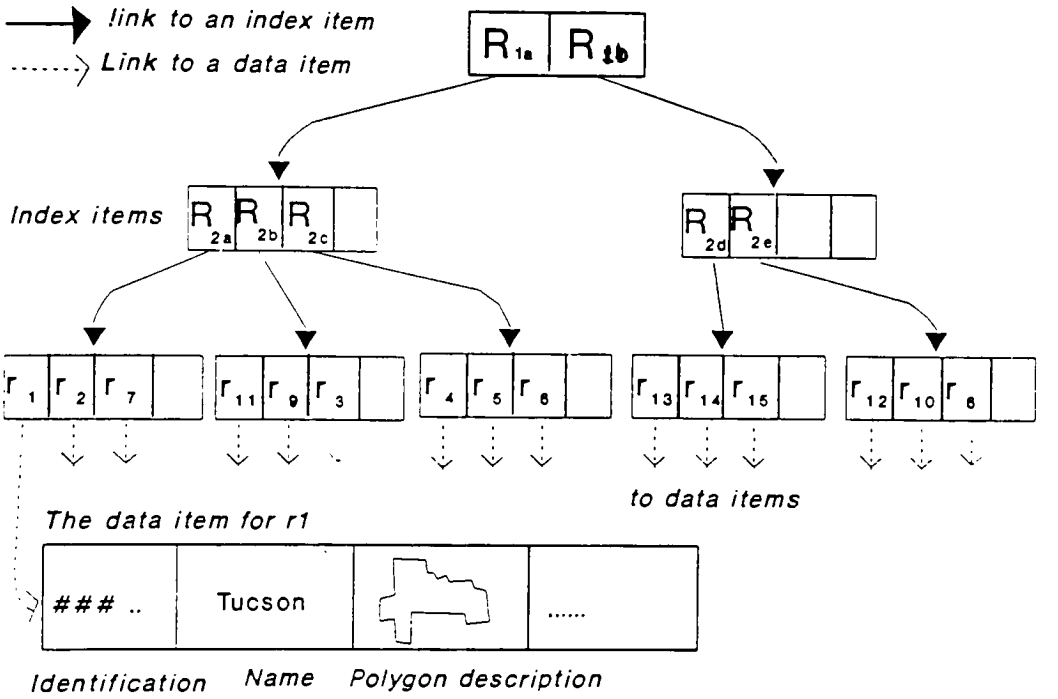


Figure 16: Corresponding R-tree

tion algorithm for the node splitting operation can affect the performance of an R-tree. The selection algorithm selects the entries that should be placed in each of the two nodes during a node split. The goal of the selection algorithm should be to minimize the area coverage of the resulting nodes, which will provide better pruning during search and to minimize the intersection area between the two nodes so as to minimize collision during search. Guttman [27] described two practical selection algorithms, a quadratic-cost algorithm and a linear-cost algorithm. The quadratic-cost algorithm runs in time quadratic in B and linear in the number of dimensions. The linear-cost algorithm runs in time linear in B and the number of dimensions.

Roussopoulos and Leifker [43] presented a packed R-tree algorithm that packs the R-tree so that the number of entries at each node will be as close to B as possible and also minimizes the area of coverage during node splits. However, the algorithm builds a static R-tree only. A static R-tree structure is not amenable to dynamic updates.

To update a data item in an R-tree requires time complexity proportional to $\log_b n$ in the worst case, where b is the order of the R-tree and n is the number of data items. Increases in b increase the logarithmic base and can lead to savings in the costs of insert and delete operations. (There are, however, other constraints on the practical value of b.) The search time is good because comparison involves simple determination of intersection between rectangles and large area pruning often occurs, resulting in the need for examination only of areas near the query region.

The structure of the R-tree enables the efficient rapid range searches that often occur in geographical data processing. Its having a B-tree like structure also maximizes secondary storage utilization when large indexes of data items are involved. This structure also contains some other desirable properties, such as shift invariance and data abstraction from actual data item representation, making it an attractive data management tool for a GIS.

In GIS analysis, the area of study often is not known at the time of data compilation. Frequently there is a need to study areas that are not demarcated by predefined boundaries such as township or municipality lines. Therefore the data structure that stores the data must have the flexibility to efficiently extract those areas within the study area. R-trees provide such flexibility by allowing rapid retrieval of all relevant data records within a defined region and efficient access to secondary storage.

When adopted for GIS data representation, the data items stored in an R-tree can be of any form: raster file, polygon, point, line or even other hierarchical structures such as quadtrees, chaincodes or other R-trees. This means that an R-tree can be easily implemented on any of a variety of existing systems based on different data structures. Figure 15 depicts a map of the State of Arizona with the major cities shown as data items, along with their minimum bounding rectangles. Figure 16 shows the corresponding R-tree, with the cities entered as data items. As indicated in figure 16, the data item can be a relation tuple in a relational database, showing the name and polygon descriptions as well as other attributes of the geographic object. The data items could be a quadtree or

an R-tree of the polygon shape for the cities, run-length encoded raster files for the cities, or any other type of representation.

2.4.1.2 R-tree for Spatial Analysis

Boolean operations on geographic data can be evaluated efficiently using R-trees. Because an R-tree is a hierarchical structure and data items have their bounding rectangles stored in the R-tree, it is possible to perform union and intersection fairly easily. Instead of calculating unions and intersections of polygonal features directly, it is possible to use the hierarchical structure of the R-tree to prune off large areas and use comparison of bounding rectangles of data items for trivial rejections. Therefore, only the true intersecting polygons would need to be calculated using the Weiler-Atherton [62] algorithm for polygon intersections.

Proximity analysis such as measuring the Euclidean distance between data items and area perimeter analysis can easily be implemented using basic geometric algorithms. Windowing a section of the world is especially fast, since only clipping of data against a rectangle is necessary.

2.4.1.3 Integration with Relational Database Technology

A GIS can be thought of as a pictorial database in that most geographical data are defined in terms of images or, geometric shapes or entities. This means that GIS implementations using relational databases have requirements similar to those of pictorial database systems.

The application of relational database to pictorial information systems has been well studied [13]. QPE [12] is a query language based on Query by Example (QBE) that was implemented to access imagery and extracted map information from a relational database. Most relational databases are alphanumeric based, and their capability needs to be extended to handle pictorial database queries. The comparison operators needed to handle a pictorial database are different from those used in strict alphanumeric applications, such as accounting or inventory information. This means that a pictorial database must have an extended set of operators that are tailored to handle queries on geometric entities such as regions, line segments and points. Some examples of these operators are AREA for the area coverage of a particular region, LENGTH the length of a given line, and DISTANCE the distance between two points [12]. The use of a relational database coupled with an R-tree has been proposed in Rousopolous [43], which describes a query language called PSQL that uses SQL like syntax to access information from a geographic database.

It might be reasonable to ponder, why do we need to use a relational database in a GIS? What are the benefits? If the geographic database is relatively small and static, using a relational database might not be necessary. But if the size of the database becomes large and dynamic then the relational database will be very much needed to handle the multitude of applications that must be performed on large amounts of data. Geographic data such as a region definition must have an alphanumeric symbol or definition associated with them if they are to be useful and valid. Without a specification of what the region represents or where it is located in the real world, the geometric region has no meaning or validity. It

is the combination of geometric specification and real world identification that makes geographic data valuable. Relational databases enable a GIS to make the connection between the point vertices of geographic objects and their real world correspondence in an efficient and structured manner.

An example of an R-tree and relational database linkage is shown in figure 16. The R-tree of a group of city polygons is shown with one of the data items, r1, in a leaf node as a tuple in a relational table.

2.4.2 R^+-Trees and Cell Trees

R^+-Trees are a variant of the R-tree proposed by Sellis, Roussopoulos and Faloutsos in [60]. The R^+-tree very closely resembles an R-tree but the rectangular regions of non leaf nodes of the R^+-tree do not overlap, and R^+-tree relaxes the fill criteria of the R-tree, i.e., each node can have fewer than the minimum number (2) of entries. The non-overlapping property of R^+-trees means association is made between each rectangular region and all the other bounding rectangular regions with which it intersects. Thus several paths to the same data item in a leaf could occur. This necessitates that the height of a R^+-tree will be higher than that of a corresponding R-tree.

The rationale for this structure is that elimination of the overlapping regions will minimize disk access during search. Empirical results in [60] show that disk access during region searching can be minimized by elimination of region overlaps. This is not surprising since we know this must be true from the

structure of the R-tree. Elimination of overlaps means collisions are minimized, so disk access must decrease. However, as we stated an R^+-tree does not have the minimum fill criteria which keeps the R-tree balanced as in a B-tree. This is a severe penalty because B-tree performance is no longer guaranteed. Therefore, it is questionable whether R^+-trees are truly superior to the conventional R-trees. More theoretical and empirical analysis should be conducted before we can determine whether an R^+-tree is truly better than an R-tree in all phases of performance.

A similar structure to the R^+-tree is the cell tree of G$\ddot{\text{u}}$nther [26]. The difference between the cell tree and the R^+-tree is that non leaf nodes of the cell tree have convex polyhedra shapes as regions of coverage instead of the rectangular regions in the R^+-tree. Clearly, the more precise definition of the polyhedra shape means that only true intersections can occur during search, which should minimize disk access. However the polyhedra descriptors are not of uniform size and could necessitate being kept on more than one disk page, thus increasing disk access during searches. As with the R^+, in cell trees, B-tree performance are no longer guaranteed.

2.5 Comparison of Spatial Data Structures for GIS

The need for rapid processing of large amounts of geographic data and efficient management of geographic databases coupled with increasing availability of low cost computing, have given rise to large scale adaptation of GIS by business and

governmental agencies [10].

As a result, there are now hundreds of GIS available to the general public. In these GIS many different data structures have been adapted for representation of geographic data, but the vast majority of today's systems use one of the two earliest data structures, cellular or vector. Testifying to our statement about the ease of implementing cellular systems and the high cost of implementing vector systems, the higher end systems such as ESRI's (ARC/INFO) and Intergraph corporation's systems are vector based. The less costly systems are usually cellular-based systems such as Construction Engineering Laboratory's (CERL) Geographic Resource and Analysis Support System (GRASS). GRASS uses cellular data structure and offers many types of spatial analysis and, in addition has software for some image processing on satellite imagery. It has been adapted as the official GIS for the Bureau of Land Management (BLM) and National Park Service (NPS). The consensus among cartographers still appears to favor the vector based system due to its high quality output. But this is changing as a result of advances in raster technology. The more advanced data structures such as quadtrees and k-d trees still are limited to implementation in the academic setting. University of Maryland's GIS (QUILT) is in its fifth version, but its spread into the commercial market has been limited by its more complex structure (making it more difficult to manage) and other performance issues discussed earlier.

The success of the two simplest data structures, cellular and vector, indicates that performance is very important to the success of a data structure for a GIS. Both cellular-based and vector-based systems can still augment their performance

and functionality by adapting a higher level data structure for managing their data. Currently most vector and cellular systems continue to lack a higher level structure that would permit range search queries to be performed efficiently. For example, some systems still require users to specify the name associated with a coordinate region instead of being able to return the correct data based on absolute location.

Problems also exist on to how relational database technology can be used in a GIS environment to facilitate queries and increase system functionality. This problem is especially acute for cellular systems. Constructs to support higher level object-oriented searching also is not readily available in most GIS.

The R-tree is a data structure that is well suited to range searching. It has also been shown to facilitate relational database queries and an SQL like extension into pictorial database called PSQL, implemented in R-tree is described by Roussopoulos and Leifker [43]. Because R-tree's structures facilitate storage of non primitive data, but rather geometric objects they can provide a high level "object oriented search" [11]. Instead of pointing at relation tuple or simple cellular file at the leaf level, the data pointer can be set to be an object-identifier. The R-tree can also be easily implemented on top of either a cellular or a vector based systems. It is also very flexible in that it can be implemented in "high" or "low" resolution. This means R-tree can be used to manage very minute or primitive data such as only line segments or to higher level geographic entities such as a forest boundary. We have shown how R-trees can be used to manage cellular and vector systems in the section describing R-trees.

We have emphasized two-dimensional R-trees in our discussions, but R-trees are not limited to two-dimensional space and extension into three-dimensional space is also possible. In [29] an R-tree is used as a spatial index structure for three dimensional airspace. Based on our survey in the spatial data structures for GIS we believe that GIS can be made more efficient by adapting an R-tree as the top level data structure for the management of geographic data.

3 Boolean Operations on Geographic Data

In this section we examine a type of spatial analysis known as boolean operation on geometric objects which is fundamental to geographic information processing. We first describe how boolean operations on geographic data are useful and then present some examples of boolean operations queries.

Current approaches to solving this type of query are examined, and reasons why they are inadequate are presented. Our approach to improving the time performance of boolean operation queries is then explained. Finally, a generalized solution to facilitate efficient boolean operations on geographic data is presented.

3.1 Boolean Operation Query

A boolean operation query allows the user to derive regions that contain a specified set of attributes by formulating and executing boolean algebra types of queries. It is a very powerful technique that is used extensively by government agencies and business. An example of a boolean operations query would be "Find all low elevation areas that contain ph levels below 7.0." This would entail intersection of low elevation regions with regions that have ph level below 7.0.

Boolean operations are such an important component of geographical data analysis that some data structures, such as quadtrees have gained many adherents because of the ease of performing them on quadtrees. Nevertheless, despite

the increasing speed of computer systems, boolean operations remain a severe bottleneck in most GIS.

3.1.1 Boolean Operations: An Extended Example

To demonstrate the use of boolean operations in geographic data processing we present a more involved example.

Let A = regions containing good topsoil.

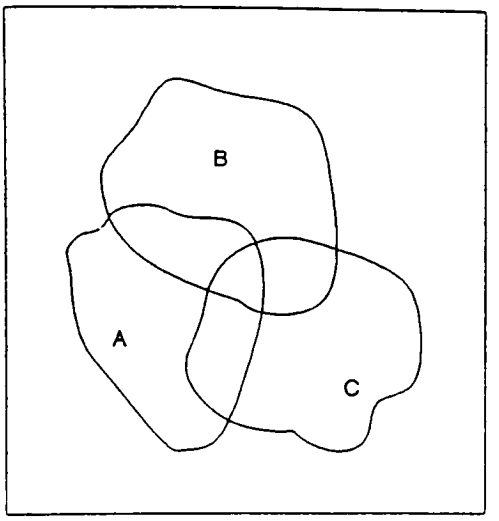
Let B = regions containing pH exceeding 7.0.

Let C = regions with good drainage.

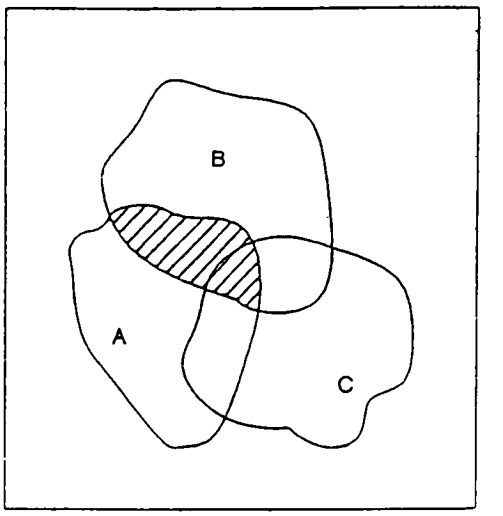
Figure 17 illustrates this with polygons A, B, and C representing regions with the attributes described. To explain the use of boolean queries involving these regions we can define the following searches:

- Find all regions that have good topsoil and pH exceeding 7.0. This translates to a boolean operation on A AND B.
- Find all regions that have good topsoil or pH exceeding 7.0. This translates to a boolean operation on A OR B.
- Find all regions with good topsoil and have good drainage or pH exceeding 7.0. This translates to a boolean operation on A AND (B OR C).

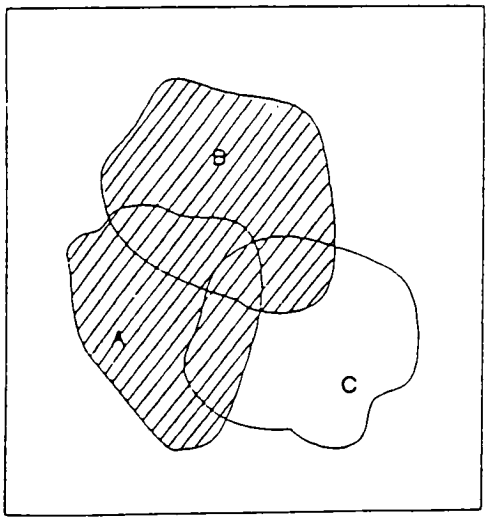
Geographic data processing frequently involves performing queries with



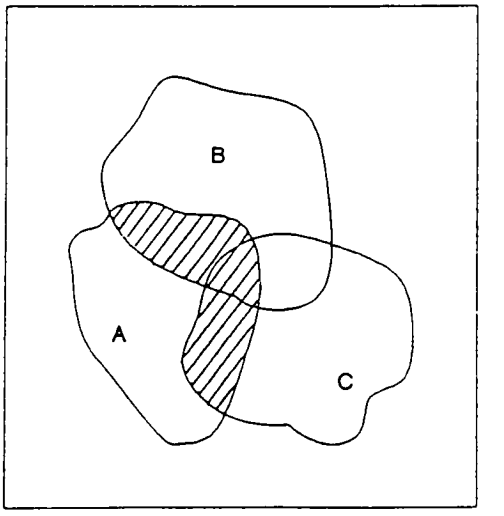
(a) Individual Sets



(b) A and B



(c) A or B



(d) A and (B or C)

Figure 17: Boolean Operation Examples

boolean operations. Business can ask question such as "Find all households in Rochester with annual income above $20,000 AND located within 9000 meters of a shopping mall." Government agencies can ask question such as "Find public land in Henrietta that is within 100 meters of Henrietta Road AND is of soil surface type concrete."

3.1.2 Boolean Operations: Current status

Boolean operations involve using the following operators on geographic data sets:

- AND or $\cap$
- OR or $\cup$
- NOT or $'$

Currently, most GIS support boolean operations using these operators. Many allow the user to formulate boolean algebraic expressions using geographic data. The problem is the efficiency of executing these user defined queries. If figure 17 represents a 1024-by-1024 pixel area and we wish to execute the simple boolean expression of "A AND B" it will require more than 1 million fetch, compare and store operations. User queries frequently involve many attribute types and can involve an extremely nested expression. This is the reason boolean operations can become a bottleneck in geographic data processing.

Previous research into this area has been concentrated in the area of data structures. A data structure such as the quadtree that exploits the homogeneity of its data can improve boolean operation performance in some situations, but it has other disadvantages. No existing data structure can guarantee good boolean operation query performance in all cases.

3.2 Current Approaches to Boolean Operations Queries

Boolean operations on geometric objects is not unique to geographic data processing. Boolean operations on geometric objects have been studied extensively in the literature due to the wealth of their applications, ranging from computer graphics, to geographic and CAD data processing purposes [47, 28, 56, 61, 62]. In VLSI applications they can be used to design printed circuit with no crossovers by detecting overlaps of conductors [56]. In computer graphics they are used in detection of polygon intersections for hidden surface removal [62].

Many solutions exist for performing a single boolean operation on geometric objects such as (A $\cap$ B) and (A $\cup$ B). We partition these solutions into two broad classes.

1. Data representation solutions - This type of solution depends on changing the data representation used to facilitate boolean operations. Quadtree is an example of this type of solution.

2. Polygonal Intersection solutions - This type of solution deals with finding

the polygonal intersections in vector data structures. The Weiler-Atherton algorithm is an example of this type of solution.

In the next section we describe examples of each of these two classes of solutions.

3.2.1 Data Representation Solutions

One way to facilitate evaluation of boolean operation queries on geometric objects is to change the data representation of the geometric objects into a form in which boolean operations can be easily computed. An example of this is the quadtree data structure. Boolean operation on data encoded in a quadtree structure can be easily performed. To obtain the results of boolean operations on two sets of data objects only requires the parallel traversal of their respective quadtrees to generate the resulting quadtree.

For example, in figure 18 two simple polygons are encoded in quadtree form. To obtain the AND (or intersection) of the two polygons we can simply follow the quadtree AND algorithm [47, 48, 51] which traverses the two input quadtrees in parallel and examines the two corresponding nodes to generate the resulting quadtree. At each node during traversal it follows the following procedure.

- If either of the nodes is white then the corresponding node in the resulting quadtree is white.

- If one node is black then the corresponding node in the resulting quadtree is set to the value of the other node.
- If both nodes are gray then the current node in the resulting quadtree is also set to gray. The processing of the sons of the current nodes in the input quadtrees will continue, but the sons of the current node in the result quadtree must be checked to determine if they are all white. If they are, then the current node in the resulting quadtree will be changed to white.

Following the AND algorithm for traversal generates the quadtree shown in figure 19, which is the resulting quadtree for the areas of intersection of the two polygons. Note that the shapes of the input polygons are very simple. This is for the sake of demonstrating our concept, this algorithm will work for very complicated polygons, even polygons containing holes.

Although it is very easy to perform boolean operations on quadtrees, the conversion to quadtree representation will require additional time and space. There also are other drawbacks to quadtrees which we have covered in our discussion of quadtrees. The most important problem associated with this approach is that it is very data dependent, and the worst case performance, that in which the data is very non-homogeneous, can often occur.

3.2.2 Polygonal Intersection Solutions

There are numerous algorithms for performing boolean operations on polygons. They are covered in many computer graphics texts [30, 19, 41] and several

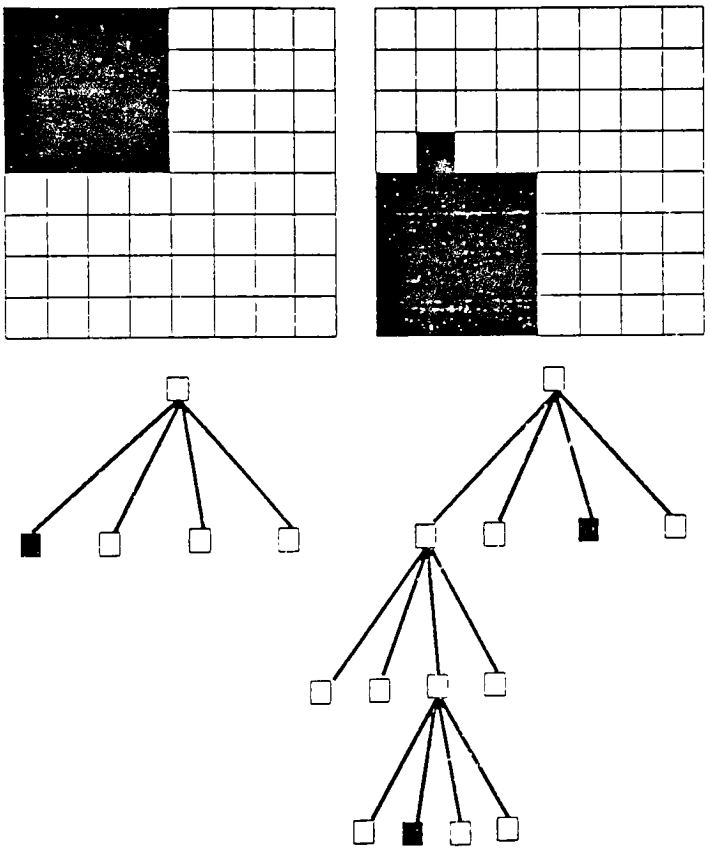


Figure 18: Two Input Quadtrees

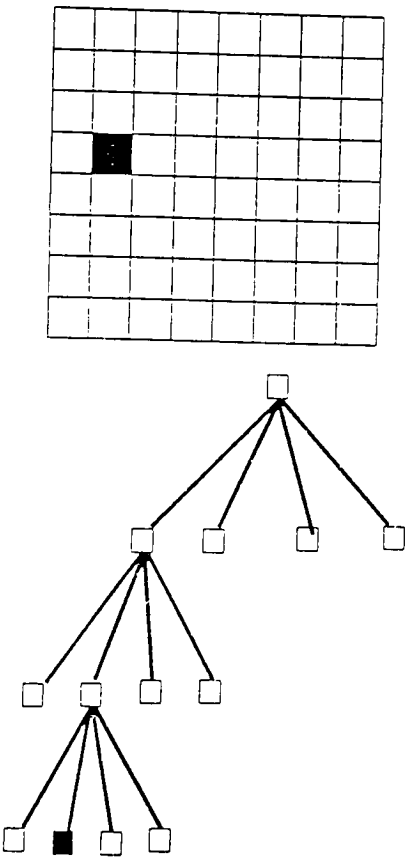


Figure 19: Resulting Quadtree from Intersecting the Two Input Quadtrees

articles [61, 56, 62, 63]. The Weiler-Atherton algorithm for finding polygon intersections appears to be the most useful because it can handle any polygon, even polygons with holes. The algorithm traces the boundaries of the polygons to determine intersections and obtain the boundaries of the resulting polygons. It requires significant time to compute because each line segment of the polygons must be examined to see if an intersections occurs. It also requires significant bookkeeping to keep track of all output polygons. The Sutherland and Hodgman polygon clipping algorithm [58] is simpler, but it handles only convex polygons and its output could contain extraneous edges. The running time of both these algorithms is large, and are $O(n \times m)$ in the worst case for intersection of two polygons with n and m vertices.

Weiler later described an interesting algorithm called the "polygon comparisons" algorithm which calculates all boolean set in one pass [63]. In the polygon comparison algorithm the union, intersection and subtractions can all be obtained by a traversal stage of the algorithm which assigns ownership information. The algorithm is based on the notion that all the output polygons' contours can be generated by merging all the input polygons contours into a graph. This is inherently true as shown in figure 20 which shows the input polygons and the graph generated by merging all their contours. It can be seen that all output polygons' contours are inherent in figure 21, and can be labeled by ownership information. The separated output polygons are shown in figure 21. The running time for this algorithm is also very large because the check for intersections must still be done. This algorithm should be used if the entire boolean set is needed, otherwise the Weiler-Atherton algorithm should be used

because it requires relatively less bookkeeping.

One technique that can be used to minimize the time required to calculate intersections is to use a bounding rectangular region to reject line segments that are very far apart. This can be illustrated by the following example. If we want to intersect the two polygons P_1 and P_2 , we can use two bounding rectangles B_1 and B_2 to facilitate our task. Instead of checking for the boundary intersections of P_1 and P_2 directly, we can use the simpler boundaries of B_1 and B_2 to determine whether intersections can occur. Only three situations can occur between B_1 and B_2 and they are shown in figures 22, 23 and 24. These are:

1. B_1 and B_2 do not intersect. This is shown in figure 22.

2. One of the bounding rectangles is wholly contained in another. For example B_2 is contained in B_1. This is shown in figure 23.

3. B_1 and B_2 have an intersection rectangle called B_{12} , as shown in figure 24.

In all three situations we can minimize the work we must do by following the following strategies. If situation 1 occurs then we know that P_1 and P_2 will not intersect. If situation 2 occurs and B_2 is contained in B_1 then we can first find the intersection polygon of B_2 and P_1 and then calculate the intersection between this polygon and P_2. If situation 3 occurs then we need only to calculate the intersections of portions of P_1 and P_2 that lie within B_{12}. By using the simpler shapes of the bounding rectangles the calculations required for boolean operations can be minimized.

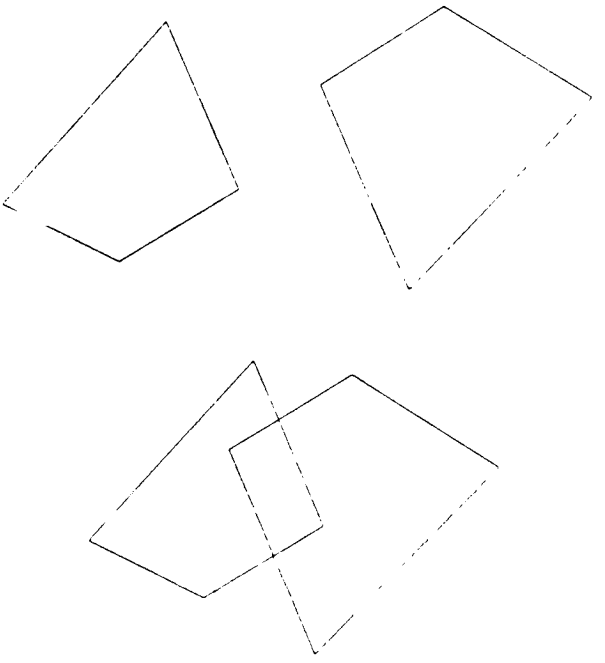


Figure 20: Input Polygons and Merged Graph

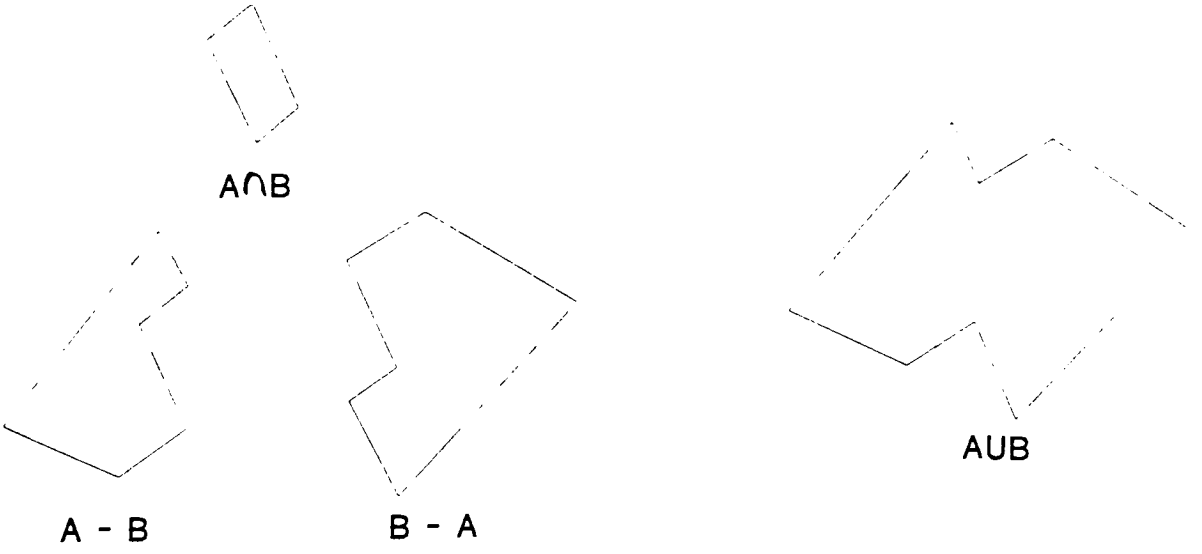


Figure 21: Output Polygons

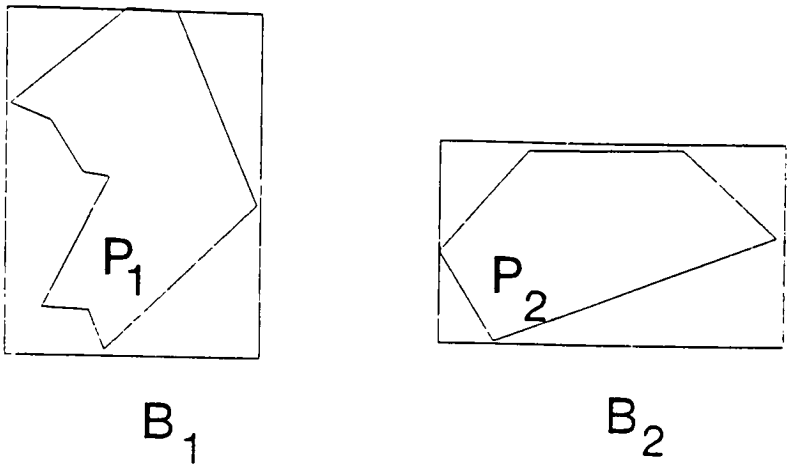


Figure 22: Hierarchical Box Test Technique (Part I)

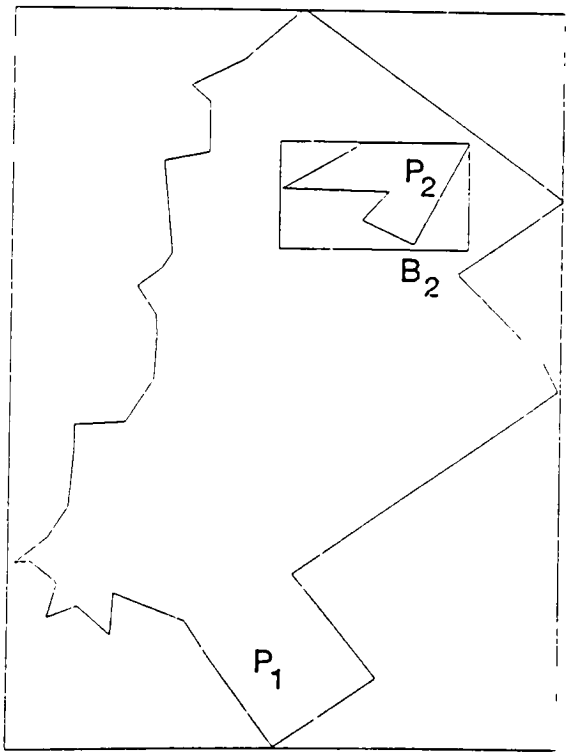


Figure 23: Hierarchical Box Test Technique (Part II)

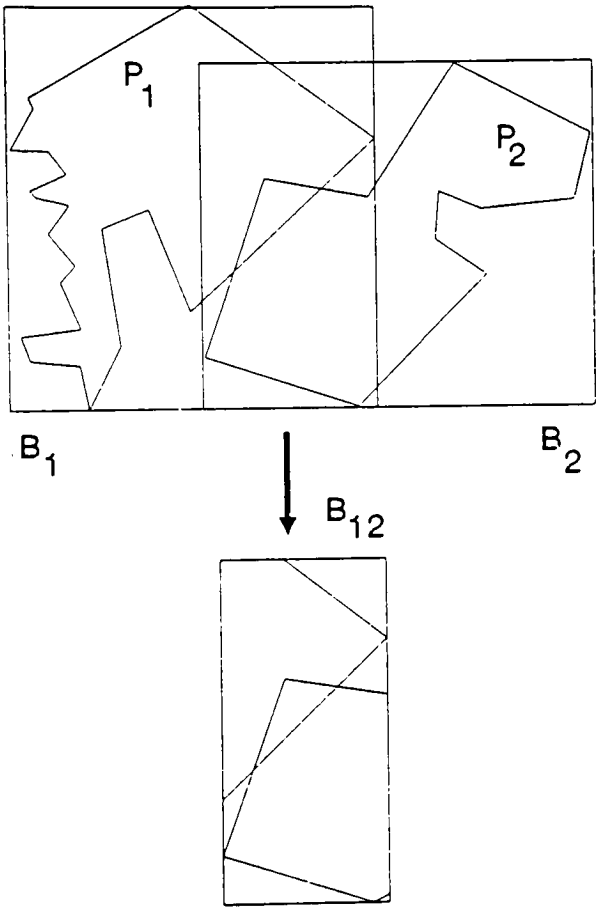


Figure 24: Hierarchical Box Test Technique (Part III)

Other techniques have also been described by Dobkin and Lipton in [15], and Shamos in [55]. These techniques use a presorting of the vertices of the boundaries to minimize calculations.

3.2.3 Summary of Current Approaches to Boolean Operations

Current approaches to boolean operations on geometric objects are effective in that they can provide the correct results, but have the disadvantage of requiring much time to perform the calculations. Both approaches described require significantly large running time to compute. Even though some techniques, such as the hierarchical box test (which has been described earlier), can be used to minimize the calculations needed, boolean operations on geometric objects still cannot be done in a timely manner. This can become a severe bottleneck if many sequences of boolean operations must be performed on multiple sets of objects.

3.3 The Proposed Approach

3.3.1 Boolean Operations: A General Solution

Attempts to devise a "perfect data structure" or a new and faster algorithm to improve the time required to perform boolean operation queries have not been successful. It is necessary to determine a new strategy to enable efficient boolean operation queries that are independent of the underlying data representation.

In most boolean operation queries the user does not simply want to obtain the result of one boolean operation on two sets of polygons. It is more likely that multiple sequences of boolean operations will be required to be performed on many sets of data. Real world problems in CAD and geographic data processing are complicated, and they frequently involve specifying more sophisticated queries requiring many sequences of boolean operations on many sets of geometric objects, and are of the form such as $(A \cap B) \cup (C \cap D) \cup (E \cap F)$ instead of just $(A \cap B)$. These types of queries are especially prevalent in geographic data modeling when finding regions that satisfy some complex set of conditions is desired. For example, to create a map of suitability classes for growing wheat it may be necessary to perform boolean operations on maps of soil surface type, slope, pH and nutrients. The resultant query will then have the implicit structure of a *boolean algebraic expression.*

Boolean expressions involving multiple sequences of boolean operations on many sets of geometric objects have received little attention in the literature. But boolean algebraic expressions have similarities to arithmetic expressions in computer programs, i.e, both are structured, have fixed syntax ,and have similar sets of algebraic laws that can be applied. This gives rise to the notion that user specified boolean algebraic expressions operating on geometric objects can be optimized in a manner similar to code optimization in compiler design. Analogous to this type of optimization is the query optimization technique used in relational database queries.

Optimization techniques have long been applied to computer programs and

are well documented. These techniques allow for the analysis of program instructions to identify redundant or unnecessary calculations. These techniques can also be applied to boolean expressions to minimize the number of boolean operations required and identify redundant calculations [3].

To evaluate this type of boolean algebraic expressions require significant computation because performing boolean operations on geometric objects requires a significant amount of computation. Despite advances in the development of new algorithms [62, 61, 56] and new data structures such as quadtrees to facilitate boolean operations on geometric data, the time required for boolean operations still is significant. This problem is magnified in the case of the evaluation of boolean expressions, because many sequences of boolean operations will need to be performed. Furthermore, current systems evaluate these expressions in a sequential manner without any preprocessing to identify redundant or unnecessary calculations.

We propose that the computation time for evaluating boolean expressions on geometric objects can be significantly reduced using strategies from the field of code optimization in compiler design. These strategies can be used in a preprocessing stage to reduce redundancy at the top level before performing the actual boolean operations on the geometric objects, leading to improved time performance.

We identify the following techniques as potentially usable for performing complex boolean operations on geometric objects.

- **Transformation of boolean expressions with algebraic identities.** A boolean operation query has its basis in set theory which has algebraic identities that can be applied to optimize the user-defined boolean operations query. Some basic properties of set theory include the commutative, the associative, the distributive, and deMorgan's laws. Successive application of algebraic transformations on an expression can generate a different, but equivalent expression. Our goal in using this technique is to minimize the boolean operations needed to evaluate a boolean expression by transformation of the boolean expression into an equivalent but more efficient expression using algebraic laws.

- **Systematic caching of intermediate results.** A caching scheme can be devised to exploit inter-query coherence, where intermediate results are saved and can be used later in the evaluation of other expressions. This can be seen in the evaluation of $(A \cap B) \cup (C \cap D)$, the subexpressions $(A \cap B)$ and $(C \cap D)$ can be saved so that later evaluation of any expression containing them could use the stored results.

- **Alias and common subexpression elimination.** A user defined boolean operation query can contain redundancies in terms of common subexpressions that are not visible at the top level. When these occur, the query can be restructured such that the subexpressions are not recomputed.

- **Short circuit code.** In evaluating an expression it may be possible to determine the results without evaluating the expression completely. This can be demonstrated by evaluating the expression "$(A \cup B) \cap (C \cap D)$."

In this example if C is the null set, the result of the expression will be null without explicitly calculating the union, A ∪ B.

It can be seen that our approach to optimization of boolean operations on geometric objects is unique and is independent of the complexities of the data, (i.e., it is not affected by how homogeneous the data are or how irregularly shaped their boundaries). The analysis is done in the symbolic domain rather than the computational geometry domain and avoids the worst case performance that can occur with many solutions that use alternative representations or require presorting of large numbers of vertices. Yet our technique can lead to significant improvements in the running time of boolean operations queries by identifying redundant and unnecessary calculations. We have basically shifted optimization emphasis from the computational geometry domain, where the evaluation time can be enormous depending on the actual complexities of the data, to symbolic evaluation in which the time required for analysis is relatively constant. Furthermore, most of the analysis can be done a priori at a compilation stage analogous to compilation of program code, thus leading to fast evaluation during runtime.

3.4 The DAG Representation of Boolean Expression

To facilitate our analysis of a boolean expression, the boolean expression is first converted into a useful structure called the directed acyclic graph (DAG). A boolean expression can be represented by a DAG in the same way as arithmetic expressions [25] and code blocks in code optimization are represented [1]. DAG

is a data structure that is widely used in code optimization because it gives information on the relationships between subexpressions. This makes it useful for analyses and transformations of expressions. Common subexpressions within an expression can be easily identified in a DAG because a node in a DAG representing a common subexpression has more than one parent.

The DAG representation of a boolean expression satisfies the following conditions.

1. The interior nodes are labeled by boolean operators $\cap$ $\cup$. The operands of an operator node are its children.
2. Leaves are *unique* identifiers representing a geometric region or a set of regions.
3. Ordering is relevant, with the leftmost child representing the first operand and the rightmost the last.

For example, the DAG for the boolean expression of ((A $\cap$ B) $\cup$ (A $\cap$ C)) $\cup$ ((D $\cap$ B) $\cup$ (D $\cap$ C)) is shown in figure 25. Derivation of DAGs from a boolean expression can be done in a way similar to the generation of expression trees from arithmetic expressions described by Aho, Sethi and Ullman in [3].

Conversion of the boolean expression into a DAG not only makes common expressions within the expression explicit, but also parses the expression. Analysis of the expression can be accomplished by traversal of the DAG, thus eliminating repetitive parsing of the expression.

3.5 Optimization Techniques Utilized

In this section we describe the various techniques used in our approach to improving the time performance of evaluating boolean algebraic expressions. As stated before these include:

1. Transformation of boolean expressions with algebraic identities.
2. Systematic caching of intermediate results.
3. Common subexpression identification.
4. Short circuit code.

3.5.1 Transformation of Boolean Expressions with Algebraic Identities

In this subsection we present a technique to minimize the total numbers of boolean operations needed to evaluate a boolean expression by transformation of the expression using algebraic identities. For example, the boolean expression ((A $\cap$ B) $\cup$ (A $\cap$ C)) $\cup$ ((D $\cap$ B) $\cup$ (D $\cap$ C)) , which requires 4 $\cap$ and 3 $\cup$ to evaluate, can be simplified into an equivalent expression ((A $\cup$ D) $\cap$ (B $\cup$ C)) which requires only 2 $\cup$ and 1 $\cap$ to evaluate. Clearly this type of transformation could lead to an expression that can be more efficiently evaluated.

3.5.1.1 Basis for Algebraic Transformation of Boolean Expression

Boolean operations use the operators $\{\cap, \cup, /\}$.

The algebraic laws [36] governing these operators are:

1. Laws of double complement.
2. Commutative laws
3. Associative laws
4. Distributive laws
5. De Morgan's laws
6. Idempotent laws
7. Absorption laws
8. Identity laws
9. Domination laws

Our technique will concentrate on using these three laws of boolean algebra concerning these operators $\{\cap \cup\}$ to optimize a boolean expression:

- The associative laws:
  $(A \cap B) \cap C = A \cap (B \cap C)$
  and
  $(A \cup B) \cup C = A \cup (B \cup C)$.
- The commutative laws:
  $A \cap B = B \cap A$ and $A \cup B = B \cup A$.

- The distributive laws:

  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

  and

  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

Both the associative and the commutative laws have been used in optimization of arithmetic expressions in code optimization and are described in various texts [3] and articles [53, 7, 24, 23, 25]. But the distributive law can also be used and it is very powerful in factorization as described by Breuer in [7]. Gonzalez and Jaja [25] describe ways in which an arithmetic expression involving only + and $\times$ operators can be transformed into an equivalent expression to minimize the number of arithmetic operations using not only associative and commutative laws but also distributive laws. This is the basis for our approach. We extend the methods described by [25] involving arithmetic expressions into the boolean algebra domain. This is valid in that the three basic laws (associative, commutative, distributive) described in [25] for arithmetic expressions similarly hold for boolean expressions. Therefore it is possible to apply this technique to transform a boolean expression into an equivalent expression which minimizes the number of boolean operations.

The technique described by Gonzalez and Jaja is not a panacea for minimization of operations in arithmetic and similarly boolean expressions, however. In particular there are some arithmetic expressions whose number of arithmetic operations cannot be reduced and others which cannot be reduced using their algorithm. Furthermore their transformation algorithm is complex and could

output an inequivalent expression. This means all output expressions must be rechecked for equivalence to the input expression. In spite of this the Gonzalez and Jaja methodology is very useful for our purposes because minimization of just one boolean operation could save not just one add or multiply instruction, as in optimization of program block, but millions of fetch and compare instructions in checking for many line intersections. This greatly increases the cost effectiveness of their algorithm in boolean expressions involving geometric objects.

Now that we have described the basis for our technique we will describe how to minimize the number of boolean operations in a boolean expression using an extension of procedures outlined in [25].

3.5.1.2 Of DAGs and Trees

We mentioned in the previous section that DAGs can be used to identify common subexpressions in a boolean expression. A tree is a DAG with no common subexpressions, i.e., none of its nodes have more than one parent. Given a DAG, D, of a boolean expression, if D is not a tree then it may contain common subexpressions. The common subexpressions in D can be eliminated if we can transform D into an equivalent tree T using algebraic identities.

If an equivalent tree, T, for a DAG, D, can be found, then T will have fewer interior nodes than D. Minimization of interior nodes leads to minimization of the number of boolean operations that need to be performed in the evaluation of a boolean expression.

This is the crux of the transformation algorithm, i.e., if the DAG representing a boolean expression can be transformed into an equivalent tree then we have achieved our goal of obtaining an boolean expression with the minimum number of boolean operations.

To illustrate this, consider the DAG in figure 25. This DAG is clearly not a tree because each of nodes A, B , C, and D has have two parents. We now examine the DAG shown in figure 26. which is the equivalent tree for the DAG in figure 25. It is a tree because none of its nodes have more than one parent. Therefore, the number of interior nodes (3) is less than the number of interior nodes (7) in the DAG of figure 25. Evaluation of the tree requires only 2 $\cup$ and 1 $\cap$ boolean operations instead of the 3 $\cup$ and 4 $\cap$ needed for the DAG. Clearly the number of boolean operations needed to evaluate an expression can be minimized if an equivalent tree can be found for a DAG of that expression.

3.5.1.3 Definitions on Boolean Expressions

Before describing how to perform the DAG transformation we must present some definitions:

- Normal Form – A boolean expression is in normal form if it is not possible to expand it using the distributive law. For example, the boolean expression

  $$(A \cap (B \cup C)) \cup (D \cap (B \cup C))$$

  is not in normal form because after applying the distributive law, the

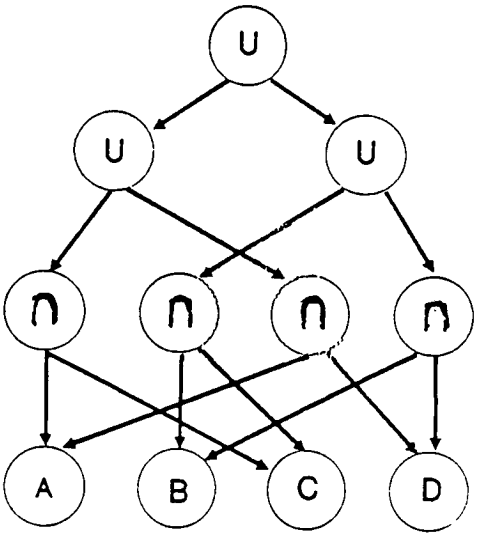


Figure 25: An Example DAG

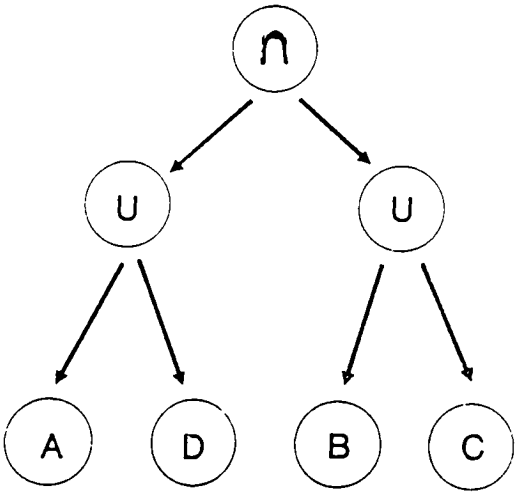


Figure 26: An Equivalent Tree

expression will be expanded to

$$((A \cap B) \cup (A \cap C)) \cup ((D \cap B) \cup (D \cap C))$$

which is in normal form.

- AND Term – A boolean expression can be written as

$$E = X_1 \cup .... \cup X_k, \text{where } k \geq 0$$

where each X_i is a leaf in the corresponding DAG or can be expressed as a result of an $\cap$ term. We call each X_i, for $1 \leq i \leq k$, the AND term of B. For example, A $\cap$ B, A $\cap$ C, D $\cap$ B, and D $\cap$ C are the AND terms of the boolean expression,

$$((A \cap B) \cup (A \cap C)) \cup ((D \cap B) \cup (D \cap C))$$

The expression

$$(A \cap (B \cup C)) \cup (D \cap (B \cup C))$$

which is not in normal form has A $\cap$ (B $\cup$ C) as one of its AND terms.

- Normal term - The AND term of a boolean expression in normal form is called a normal term.

- Left-ands of a DAG D comprise the set of leaves which are the leftmost children of the normal terms of D.

- Right-ands of a DAG D with left-ands of $\{l_i\}_{i=1}^{k}$ comprise the set of DAGs $\{d_i\}_{i=1}^{k}$ such that D is the equivalent to the DAG

$$D' = (l_1 \cap d_1) \cup (l_2 \cap d_2) \cup ... \cup (l_k \cap d_k)$$

3.5.1.4 Approach to Transformation of a DAG

The goal of this transformation is to generate an equivalent tree for a DAG representing a boolean expression. This can be done by using the divide and conquer approach outlined in [25]. The boolean expression is partitioned into a set of subexpressions which are recursively transformed into a tree. These subtrees are then combined to generate the tree for the whole expression. Note that equivalence of the output tree to the input DAG is not guaranteed by following the procedure described by Gonzalez and Jaja [25] , which could output an inequivalent tree to the original DAG.

This is troublesome because each output tree must then be checked for equivalence to the input DAG, requiring additional time to compute. Gonzalez and Jaja reported that in order for an output tree T to be equivalent to the input DAG D the following conditions must be satisfied.

1. Each of the normal terms of D is a normal term of T.
2. D and T have an equal number of normal terms.
3. D does not contain two equal normal terms.

If all these conditions are true then T is equivalent to D. An obvious solution would be to transform both T and D into normal forms and compare their normal terms to see if they satisfy the three conditions. This could be expensive, however because transforming a DAG into a normal form could lead to an exponential growth in the number of nodes of the DAG. For example the DAG

for the expression

$$D = (x_1 \cup x_2) \cap (x_3 \cup x_4) \cap \ldots \cap (x_{2n-2} \cup x_{2n})$$

in normal form could have more than 2^n edges.

Gonzalez and Jaja describe three sets of procedures which must be applied to be certain that the output T is indeed equivalent to the input D. These include two separate labeling procedures for both T and D to check for condition 1.) A counting procedure which counts the number of normal terms in D and T to check for condition 2.) Condition 3.) is checked by first transforming D into a left-justified DAG (i.e., every $\cap$ node of such a DAG has a leaf as its left child) then checking every $\cup$ node to determine if it contains two equal normal terms. If these procedures show that the output tree T is not an equivalent tree to D then both the time expended to generate the tree T and the time expended to check for their equivalence are wasted and no gain can be realized since T cannot be used.

Because our objectives are different from theirs we do not have the rigorous need to obtain a tree from a DAG. We need only to reduce the number of boolean operations in a boolean expression and do not need to find the absolute minimum number of boolean operations in a boolean expression. For our purposes, common subexpressions can be tolerated because identifying them enable us to compute them only once. How this will be useful will be covered in more detail when we discuss our handling of common subexpressions.

We present a transformation procedure which, when it can be done easily

will generate a tree from a DAG , and always guarantee that the output will be equivalent to the input. This will eliminate the need to check for equivalence between input and output. But the procedure does not guarantee that the output will always be a tree. Because of this, the output could also be a DAG, so we must compare the number of boolean operations in the input DAG with our output tree (or DAG as the case may be) to determine which requires more boolean operations to evaluate. Therefore we save the non trivial amount of time needed to determine if the output tree is equivalent to the input DAG, but at the cost of missing some (more difficult to detect) cases when an equivalent tree can be generated.

3.5.1.5 Transformation Procedure

The actual transformation can be done by taking the following steps, using the definitions defined in the previous section. Note that the output of this transformation could also be a DAG, in which case we would compare the number of boolean operations in the input with that of those in the output and use the one requiring the fewest number of boolean operations.

1. Obtain the set of left-ands and their corresponding right-ands.
2. Combine all left-ands with empty right-ands with $\cup$ and set it to t'.
3. Recursively invoke transform on each of the right-ands.
4. Each of the trees T generated from the right-ands can be written as

$$T = t_1 \cap .... \cap t_k, \text{where } k \geq 1$$

and each t_i is a leaf or a subtree with a $\cup$ root. Partition the trees generated from the right-ands into sets of trees where trees belonging to the same set have overlaps, i.e., have t's which are equal.

5. Combine the left-ands with the right-ands with the $\cap$ operators.

6. For each (left-ands $\cap$ right-ands) within a set that have overlaps. The parts without overlaps are combined with the $\cup$ operator and then are $\cap$ with their common overlaps. The results therefore will be a series of terms which are $\cap$ with their common overlaps.

7. Finally generate the resulting tree t by combining t' with the above terms for each set with the $\cup$ operators.

We give some examples of the transformation process.

Example 1:

Given the boolean expression

$$E = ((A \cap B) \cup (A \cap C)) \cup ((D \cap B) \cup (D \cap C)$$

The set of left-ands are $\{A, D\}$ with the corresponding set of right-ands { (B $\cup$ C), (B $\cup$ C)}.

At step 3 the recursive calls to transform on the right-ands return (B $\cup$ C) for both right-ands.

At step 5 the left-ands are combined with the right-ands to obtain

$$\{(A \cap (B \cup C)), (D \cap (B \cup C))\}$$

At step 6 the non overlapping parts are combined to generate $(A \cup D)$ and then $\cap$ with the common overlap to generate

$$(A \cup D) \cap (B \cup C)$$

Since there is only one set, the result is

$$(A \cup D) \cap (B \cup C)$$

Which is an equivalent tree for the DAG

$$E = ((A \cap B) \cup (A \cap C)) \cup ((D \cap B) \cup (D \cap C))$$

Example 2:

Given the boolean expression

$$E = (((A \cup (B \cap C)) \cap F) \cup (B \cap ((C \cap E) \cup (D \cap F)))) \cup ((A \cup (B \cap D)) \cap E)$$

The set of left-ands are (A, B) with corresponding right-ands of

$$\{(F \cup E), ((C \cap (F \cup E)) \cup (D \cap (F \cup E)))\}$$

At step 3 of the procedure the recursive call to transform on the right-ands of A generates $(F \cap E)$ and for right-ands of B generates

$$((C \cup D) \cap (F \cup E))$$

At step 5 of the procedure the left-ands and right-ands are combined with the $\cap$ to obtain the terms $(A \cap (F \cup E))$ and

$$(B \cap ((C \cup D) \cap (F \cup E)))$$

At step 6 the non overlapping parts are combined with the $\cup$ operator to obtain

$$((B \cap (C \cup D)) \cup A)$$

which are $\cap$ with their common overlap to obtain the result

$$((B \cap (C \cup D)) \cup A) \cap (F \cup E)$$

Which is an equivalent tree to

$$E = (((A \cup (B \cap C)) \cap F) \cup (B \cap ((C \cap E) \cup (D \cap F)))) \cup ((A \cup (B \cap D)) \cap E)$$

3.5.2 Systematic Caching of Intermediate Results

In many geographic applications the intermediate results of boolean operations can be reused. There currently is no systematic caching method to facilitate the reusing of precomputed results. We present an efficient technique for storing and retrieving boolean expression results.

In the preceding section we have shown that a boolean algebraic expression can be represented in a tree. Aho [1] describes an efficient technique for a tree matching algorithm using an extension of the Aho-Corasick [2] multiple-keyword pattern matching algorithm.

Boolean algebraic expressions can be used to construct a trie which can be converted into a pattern matching automaton that can then be used for the storing and retrieval of boolean expression results. The pattern matching automaton is similar to a finite state automaton with a start state and a set of accepting states. In our application the accepting state would point to results of previously computed boolean algebraic expression results. The pattern matching automaton in figure 27. demonstrates this concept, with accepting states of 3, 4, 7, and 8 pointing to locations where results for their path string are stored.

However, a modification to our adaptation of the Aho-Corasick algorithm is needed. In that algorithm the order of the keywords is relevant, i.e., anagrams cannot be accepted. Therefore, even though subexpressions such as (A $\cap$ B) and (B $\cap$ A) are the same according to the commutative laws they will not be treated as the same expression by the algorithm. To remedy this type of behavior we make the following modification. In accordance with both associative and commutative laws, if subexpressions can be sorted then they are first sorted in a uniform fashion such that identical subexpressions differing only by their ordering will be the same; i.e., (A $\cap$ B $\cap$ C) will always be equivalent to (A $\cap$ C $\cap$ B) because the latter expression will first be sorted to appear to be the same prior to building of the string matching automata.

Regarding the use of this technique, it should be noted that Systematic caching of intermediate results should be used judiciously because the results of boolean operations queries can have large space requirements. One possible application would be to store the results within a session, since it is likely that

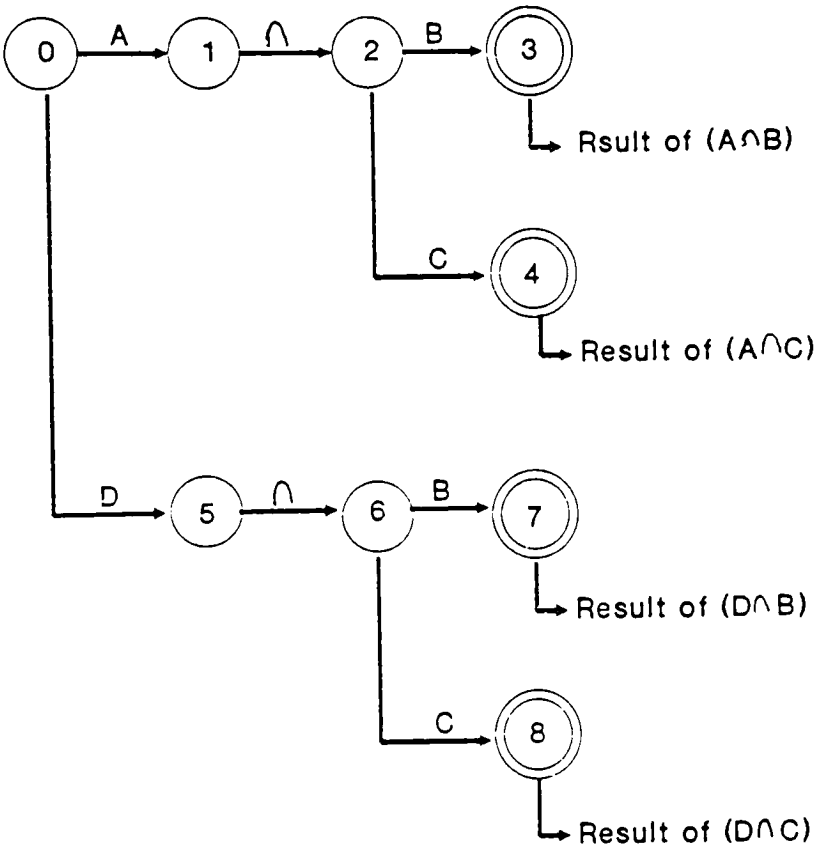


Figure 27: Automaton for Matching Boolean Expressions

the user would have high probability of referencing results recently computed (locality of reference).

3.5.3 Common Subexpression

This section addresses the problem of common subexpressions which can occur in a given boolean expression. As stated earlier, some DAGs of a boolean expression cannot be transformed into a tree. This means common subexpressions cannot in all cases be eliminated. But they can be easily detected during generation of the DAG from an expression. If the common subexpression is not a leaf then it must be tagged so that it will not be recomputed. This can be done by tagging the root of the subexpression.

There is a problem with some common subexpressions which are not explicit. While common subexpression which are explicit can be easily identified because their root node have more than one parent, implicit common subexpressions are much harder to detect because they do not exhibit such a telltale sign. They occur because they are hidden by the ordering of the terms in an expression. Aho et al. [3] describe instances in which this can be detected by using the associative and commutative laws. For example, the expressions A $\cap$ B and B $\cap$ A are the same expression under the commutative law. This is a situation similar to the one discussed in the previous section on caching of intermediate results and can also be solved using a pre-sorting of selected terms.

3.5.4 Short Circuit Evaluation

In evaluating an expression it may be possible to determine the results without evaluating the expression completely. If either of the expressions (A $\cup$ D) or (B $\cup$ C) is the NULL set, then the results of the expression (A $\cup$ D) $\cap$ (B $\cup$ C) will also be NULL.

This type of analysis can easily be performed using the DAG structure described earlier. To perform short circuit evaluation on a DAG structure requires a bottom-up strategy that examines the leaves of the DAG to determine the leaves that are the empty set. If any leaf is NULL then it can propagate up the tree. If the propagation reaches the root of the DAG and the root is a $\cap$ node then the expression will be NULL.

Short circuiting techniques can be a cost effective way to cut down on the computing needed compare to a sequential evaluation of a boolean expression, because unnecessary boolean operations within a boolean expression can be easily eliminated.

We present an algorithm which given a boolean expression in DAG representation, can be identified and exploit short circuit situations. The general strategy of the algorithm is very simple. The DAG is traversed in a depth first traversal until a leaf is reached, and its value is examined. If the value is NULL then a potential short circuit has been identified. We then ascend a level and examine its parents. All parents which are $\cap$ nodes are set to NULL and all their parents are pushed onto the reverse queue. This ascension continues until either

the root is reached or the reverse queue is empty. If the root is reached and it is a $\cap$ node then the procedure returns and the result of the whole expression is NULL. If the root is reached and it is not a $\cap$ node then the traversal downward is continued by processing the queue.

Two procedures are defined:

- short-circuit - This procedure traverses the DAG from the root in a pre-ordered traversal. When a leaf is reached it is examined to determine if its value is NULL if it is, then the procedure reverse-traverse is called on this leaf to ascend toward the root. Note that this procedure returns the value NULL when it has determine that the whole expression has been short-circuited, in which case short-circuit returns NULL.

- reverse-traverse - This procedure traverses the DAG from a leaf toward the root. This procedure will ascend the DAG from a leaf and continue exploring paths that only have $\cap$ nodes, and each of these nodes is set to NULL. Anytime the procedure encounters a $\cup$ node along a path, that path will not be explored further. If the root is reached and is found to be a $\cap$ node this procedure will return a NULL. In all other cases it returns TRUE.

After processing using this short circuit algorithm, subtrees of the DAG representing unnecessary calculations will be set to NULL.

3.6 Summary

Prior to introduction of the techniques presented here, boolean expression queries involving geometric objects had to be formulated very carefully because evaluation time increased significantly in direct proportion to an increase in the nesting level and number of boolean operations in the boolean expression. We believe this should not be the case. The problem with sequential and non-optimizing ways of evaluating boolean expression queries is that significant opportunities for minimizing calculations are not being exploited. In our approach the boolean expression is evaluated in a parallel fashion, with many subexpressions evaluated first to determine if any speed up can occur by eliminating unnecessary or redundant calculations. Our techniques are also very well suited to a parallel processing environment in which the granularity of the hardware architecture could determine the size of the subexpressions into which the expression could be partitioned.

We have presented ways in which many redundancies can be identified and used to improve the efficiency of evaluating boolean expressions. Our techniques are such that, the more nested and complex boolean a expression is the more likely it can be optimized. This is intuitively true because as the number of boolean operations in an boolean expression increase the more likely it is that one of the boolean operation will have an dominant effect on the final result.

Our techniques also frees the user of the system from being concerned with how to formulate the boolean expression for efficiency, because our techniques

```
short-circuit(t)
/** preorder descent. **/
/** Identifies the short circuits within t to determine unnecessary calculations **/
1. tag t.
2. current ← t.
3. Push all children of t onto queue.
4. While queue is not empty
      begin
        current ← pop a node off the queue;
        if current is not tagged
        then
          tag current;
          if current is a leaf
          then
            if current is NULL
            then
              code ← reverse-traverse(current);
              if code = NULL
              then
                return NULL for the expression;
              end-if;
            end-if;
        else
          push all children of current onto the queue;
          end-if;
        end-if;
      end.
5. Return TRUE
```

reverse-traverse(l)

/** Traverse the DAG from the leaf l going toward the root **/

/** If the root is reached and it is a ∩ node then NULL is returned **/

/** to signify that the whole expression is set to NULL. Otherwise a TRUE is returned **/

1. tag l.
2. current ← l
3. Push all parents of l onto queue.
4. While queue is not empty

```
    begin
      current ← pop a node off the queue;
      if current is not tagged
      then
        tag current
        if current is the root
        then
          if current is a ∩ node
          then
            return NULL;
          end-if;
        else
          if current is a ∩ node
          then
            push all parents of current onto the queue;
            set current's value to NULL;
          end-if;
        end-if;
      end-if;
    end.
```

5. Return TRUE.

will identify a non-efficient expression and replace it with one that is efficient and yet equivalent. This is analogous to the code optimizing compilers which can detect non efficient code and replace it with code that can be more run more efficiently. This will lead to faster development time for the user. Imagine a situation in which no optimizing compilers existed and programmers had to analyze their code to identify all the inefficiencies in their code. Yet this is what is expected from many users in geographic and CAD applications.

3.7 Implementation Issues

The generalized technique describe here can be applied to system using any number of data representation schemes, such as raster, vector, quadtrees, etc. Most of the techniques can be coded by modifying standard data structure subroutines.

The short circuiting code procedures should be the most cost effective technique because they can produce the result of evaluating boolean expression in many instances without performing any boolean operations. The analysis of boolean expression can be done in a type of compilation stage which does the analysis on the expression to build the data structure and perform any transformation that may be applicable. Then specific data sets could be run by calling this "compiled code."

4 Concurrency of Operations on R-trees

In a multi-user environment, whether the geographic database is at a centralized location or distributed over several locations, it would be beneficial to provide concurrent accesses to the database by multiple users needing to perform queries, analyses and updates on the geographic database. Furthermore, concurrency control in R-trees is an essential issue that must be addressed in performing distributed (parallel) computing on R-trees. This section describes the extension of R-trees to allow locks-controlled concurrent operations and presents algorithms to perform concurrent operations.

Concurrency in B-trees has been extensively studied, and many solutions has been discovered [44, 5, 17, 35, 33, 32]. Samadi [44] proposed the first solution for concurrency in B-trees in 1976. Bayer and Schkolnick [5] also presented three solutions in 1977. Because R-trees are a special case of B-trees, methods for concurrency in B-trees can easily be extended to R-trees.

4.1 Locking Mechanism Definitions

To enable correct concurrent accesses, it is necessary to utilize a set of locks on the nodes of the R-tree. A compatibility and convertibility graph (CCG) describes the relationships among the different lock types. All the solutions described in [5, 17, 33] used three types of locks: a read lock (r-lock), a write lock (w-lock) and an exclusive lock (e-lock). The CCG in figure 28 describes the

relationships among them. In a CCG, the vertices represent the lock types, and the edges are used to describe the compatibility and convertibility relationships among the lock types. Solid edges represent full compatibility, so any process holding an r-lock on a node would allow other processes to hold an r-lock or a w-lock on that node (shown in figure 28). An e-lock is incompatible with both the r-lock and the w-lock and is placed on a node to prevent other processes to access that node. An undirected broken edge represents full compatibility and convertibility between the lock types (not shown in figure 28). A directed broken edge represents convertibility from the source lock type to the destination lock type (shown in figure 28). Any process holding a w-lock on a node may convert it into an e-lock.

4.2 Type 1 and Type 2 Solutions

Kwong and Woods [33] partitioned the solutions to the B-tree concurrent access problem into two broad types: Type 1 and Type 2 solutions. In Type 1 solutions, the entire scope of an updater must be locked during restructuring, so no other updater can access it. In Type 2 solutions, only the nodes that could be affected by the restructuring are locked, leading to a scope that is more local. Theoretically, type 2 solutions should provide slightly greater concurrency, but at the cost of an extra level of concurrency control and a more complicated implementation.

The R-tree extension enabling concurrency will be based primarily on the

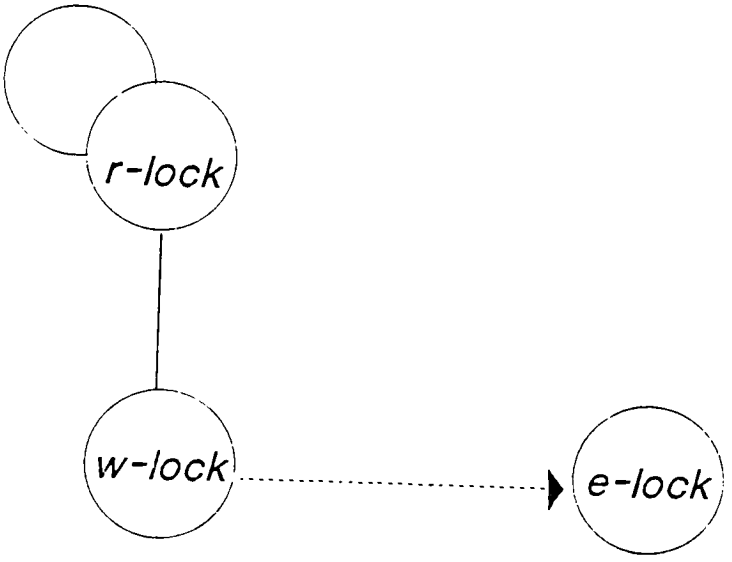


Figure 28: CCG Describing Relationship Between the Three Lock Types

third solution, proposed by Bayer and Schkolnick [5], which is a Type 1 solution. Kwong and Woods [33] presented an improved Type 1 solution based on solution 3 of Bayer and Schkolnick [5] and used a new structuring technique called "side branching" for greater concurrency. However, R-trees do not have the same concept of adjacency as B-trees, so "side branching" could not be effectively utilized.

As in [5, 33] we define three locks as needed for our concurrency control protocol, namely the r-lock, the w-lock, and the e-lock. Processes can perform three types of indivisible locking operations: lock, unlock, and convert. But the granting of these operations is subject to the defined constraints in the CCG. If permission is not granted for a locking operation, the process is put into a queue for the associated node. This protocol is enforced by a process control scheduler. We now present algorithms for allowing the concurrent search, insert, and delete operations on an R-tree.

4.3 Algorithms Descriptions

4.3.1 Search Algorithm

The search procedure uses only the r-lock for locking the nodes it is searching. As shown in the CCG for our solution, the r-lock cannot be executed on a node which is e-locked but can be done on a node that is w-locked. A recursive search algorithm using locks is described below.

```
SEARCH(t, r)

/** Recursive descent. **/
/** Return all records in tree t that overlaps search region r **/

1. Place r-lock on t.
2. If t is not a leaf.
     then
       place r-lock on children of t whose region intersects r and unlock t;
       invoke search on all children of t whose MBR intersects r;
     else
       return all entries in t whose rectangle intersects r.
3. Release all locks.
```

An alternative search algorithm using iteration is also presented. It traverses the tree in a preordered traversal and eliminates the duplicated read locking in the recursive algorithm.

4.3.2 Insertion Algorithm

The notion of safeness of an R-tree node is derived from the generalized concept of safeness for a B-tree node in a concurrent operations environment. A node in an R-tree is insertion safe if it has less than $B - 1$ entries, and is deletion safe if it has more than $b + 1$ entries, where B and b are the maximum and minimum numbers of entries in a non-root node.

SEARCH(t, r)

/** preordered descent. **/

/** Return all records in tree t that overlap search region r **/

1. Place r-lock on t.
2. current ← t
3. Push all children of t onto queue.
4. While queue is not empty

```
    begin
      temp ← pop a node off the queue;
      place r-lock on temp;
      release r-lock on current;
      current ← temp;
      if current is leaf
      then
        report all records in current that intersect r;
      else
        push all children of current whose region intersects r onto the
        queue;
    end.
```

5. Release all locks.

Some changes had to be made to the original concurrent B-tree insert algorithm to enable it to work correctly on R-trees. The problem lies in the global scope of the rectangular regions of the R-trees. A B-tree insertion in the leaf node does not affect the keys in the upper levels of the tree above the deepest insertion safe node (DISN). But in an R-tree, changing the leaf node will always propagate the modifications on all nodes' rectangular regions up the tree even beyond the DISN. The solution lies in placing a w-lock on each node as we descend the tree and when necessary changing the rectangular region to cover the data item. As soon as we encounter an insertion safe node we can immediately release all locks on the ancestor of the safe node. When we get to the leaf node, if it is full, we need to convert all w-locks to e-locks up to the DISN, then make the insertion and propagate the modification of rectangle region up to the DISN.

4.3.3 Deletion Algorithm

The R-tree deletion algorithm also encounters the same problem of the global scope of the rectangle regions described in the previous section. However, the solution is more difficult because in the R-tree delete algorithm, multiple paths down the tree could occur and no determination can be made on whether a node's rectangle region is truly caused by the MBR of the data item we are trying to delete. Therefore we cannot adjust the rectangle region of nodes encountered during the descent. So the R-tree delete will always have to propagate the rectangle region modifications caused by the deletion all the way up to the root, but only after the deletion occurs in the leaf node. A path down the R-tree

INSERT(rec, t)

/** Insert record rec into tree t **/

1. Place a w-lock on t.
2. current ← t.
3. DISN ← current.
4. While current is not a leaf node

    begin

        if current's region needs adjusting because of rec

        then

            convert current's w-lock into e-lock

            adjust current's rectangle region of coverage to cover rec's region;

            convert current's e-lock into w-lock

        place w-lock on the child whose region will need least enlargement to include rec;

        call this child favorite;

        current ← favorite;

        if current is insertion safe

        then

            DISN ← current;

            release all locks on current's ancestors;

    end.

5. If rec is in current

        then

            release all locks;

            return

        else

            ascend to DISN and convert all w-locks to e-locks;

            insert rec into current;

            propagate node splits and region modifications up to DISN.

6. Release all locks.

would have to be e-locked before the actual deletion. The delete algorithm uses a queue to keep track of potential nodes that must be explored. It traverses the tree in a preorder traversal, w-locking the nodes in its path until a leaf node is encountered. Then, if the data item is in the leaf, all w-locks will be converted into e-locks, the data item will be deleted, and all node splits and region modifications will then propagate up the tree.

4.4 Examples

This section contains some examples using the search, insert and delete algorithms described earlier.

Example 1: Given the R-tree shown in figure 29, search for records in search region r.

After step 1, the root of the R-tree shown is r-locked. At step 3 the nodes R_{1a} and R_{1b} are put into the search queue. At step 4 the node R_{1a} is popped off the queue and r-locked, the r-lock on the root is released. Since only R_{2a} intersects the search region r, it is put on the search queue. Finally, node R_{2a} is popped off the queue and r-locked, releasing the r-lock on R_{1a}. Because R_{2a} is a leaf node all record items it points to that intersect search region r are returned. This means record r_1 is the result of the search. Step 5 releases all locks on the tree.

Example 2:

Delete(rec, t)

/** Delete algorithm **/

/** Delete a data-item rec from R-tree t **/

1. Place a w-lock on t.
2. current ← t.
3. Push all children of t onto the queue.
4. While queue not empty

```
    begin
      temp ← pop a node off the queue;
      place a w-lock on temp;
      unlock current;
      current ← temp;
      if current is a leaf
      then
        if rec is in current
        then
          ascend up the tree and convert all w-locks to e-locks;
          delete rec from current;
          propagate node elimination and region modifications up the tree;
          set queue to be empty;
          report rec deleted;
        else
          push all children of current whose region entirely covers rec's
          region on the queue;
    end
```

5. Release all locks.

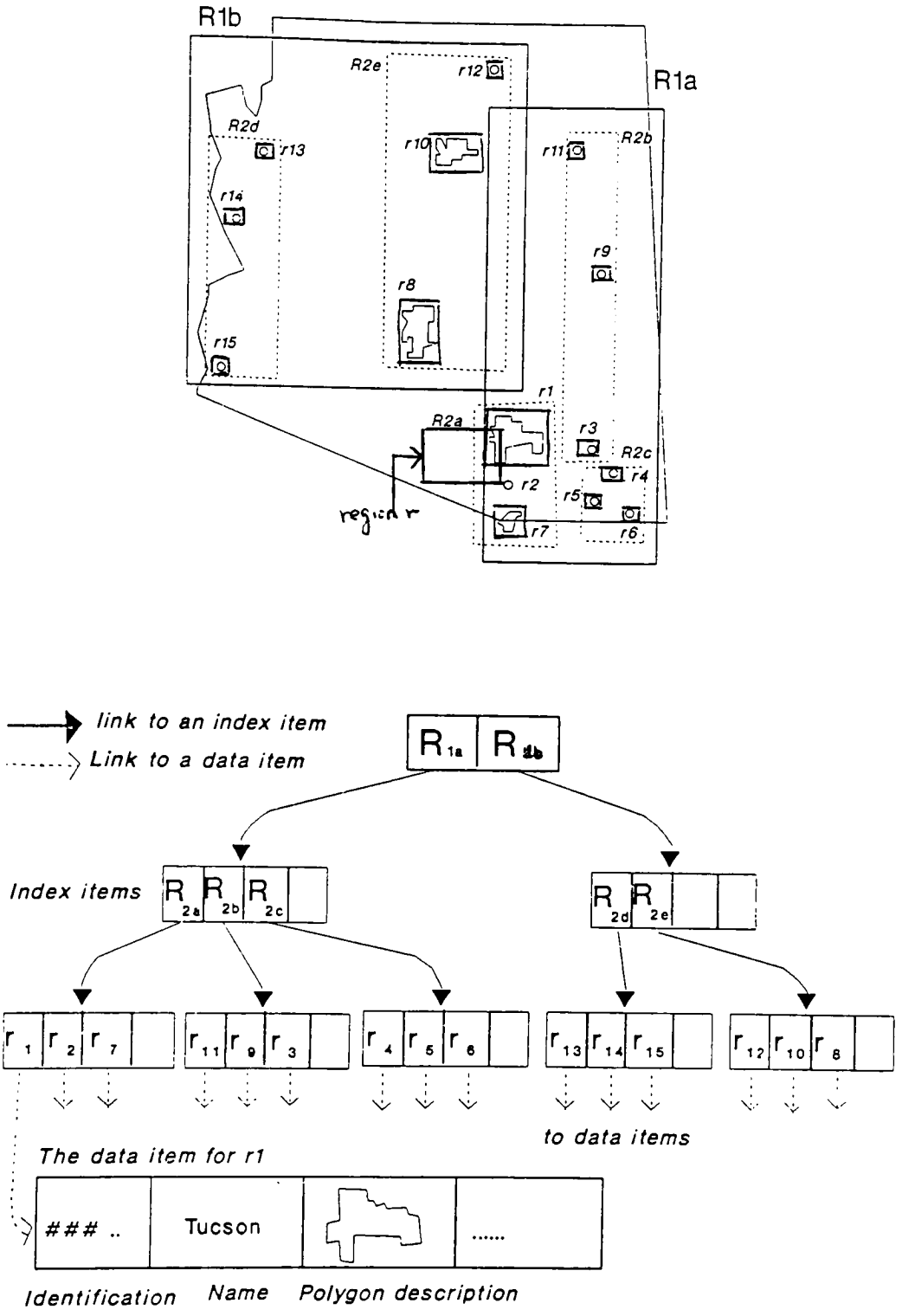


Figure 29: Example R-tree

Given the R-tree shown in figure 29, insert record item r into the tree. Note that the search region is the same region used in the previous example, except that in this example the region denotes the bounding rectangle of the data item being inserted.

After step 3 the root of the tree is w-locked, which allows only processes searching the tree to access the root. This is due to the fact that two processes cannot simultaneously hold a w-lock on a node as dictated by the CCG for our protocol. The r-lock can be held by many processes however, as shown by a solid edge that points back to itself. The DISN (deepest insertion safe node) is set to the root of the tree. At step 4 the node R_{1a} is w-locked, releasing the w-lock on the root. Because R_{1a} has only 3 entries it is set as the DISN. The w-lock is converted into an e-lock and the R_{1a} region is modified. At step 5 the record r is inserted into R_{2a} after w-lock on R_{2a} is converted into a e-lock. Step 6 releases all locks on the tree.

Example 3:

Given the R-tree shown in figure 29, delete record item r_1 from the tree.

As in the insertion algorithm, the root of the tree is first w-locked. In step 4, the node R_{1a} is w-locked and the w-lock on the root is released. R_{2a} is then w-locked releasing the w-lock on R_{1a}. Finally the w-lock on R_{1a} is converted into an e-lock and record item r_1 is deleted.

4.5 Caveats

The concurrent algorithms described for R-tree are based on the solutions described in Bayer and Schkolnick [5]. Some correctness proofs for those solutions are described in [5]. The difficulties involved in producing formal correctness proofs for this class of solutions are well known. There also is no clear consensus on how to determine whether one solution is better than another. The degree of concurrency has been proposed as one measurement of the efficiency of a solution [33]. But again, for lack of a precise definition, "degree of concurrency" is more or less an intuitive notion, which can mean different things depending on the practitioner. In addition, the problem of error recovery in cases of system failure is seldom addressed in many of the published works on this type of solutions. This is an important issue which should be addressed. In [40] there is some discussion regarding this topic.

5 Summary and Future Directions

In this thesis we have described some issues critical to the improvement of the performance of GIS. We have surveyed many of the relevant spatial data structures used in geographic data representation. The result of this survey shows that an R-tree can be used to augment the performance of many existing GIS because of its ability to access secondary storage devices efficiently, its ease of integration with relational database technology, its support of object oriented spatial searches and efficient range searching. We have shown that an R-tree

has unique advantages over other structures, to which has been added the concurrency control mechanism for R-trees that we have defined. No other known spatial data structures for geographic data representation have this advantage.

Although practitioners in the field of geographic data processing have begun to realize the potential of the R-tree data structure, it has not yet been implemented on any existing production GIS. This is puzzling since R-trees are becoming quite well known and many references to it can be found in the literature. We do not believe this will continue to be true in the very near future.

Another important contribution of this thesis is the fundamentally different approach taken toward improving the performance of boolean operation queries on geometric objects. It should be noted that to specify correct and efficient boolean operations is not easy and usually is a task reserved for personnel who have both cartographic knowledge and some understanding of the workings of the GIS. We believe that actual implementation of our techniques on a existing GIS will significantly improve not only the performance of the system but also its user friendliness.

References

[1] A. V. Aho. Code generation using tree matching and dynamic programming. *ACM Trans. on Database Systems*, pages 491 – 516, October 1989.

[2] A. V. Aho and M. J. Corasick. Efficient string matching. *Communications of the ACM*, pages 333 – 340, June 1975.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[4] D. H. Ballard. A hierarchical representation for curves. *Communications of the ACM*, pages 310 – 321, May 1981.

[5] R. Bayer and M Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1 – 21, 1977.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, pages 509 – 517, September 1975.

[7] M. A. Breuer. Generation of optimal codes for expressions with factorizations. *Communications of the ACM*, pages 333 – 340, March 1969.

[8] P. A. Burrough. *Principles of geographical information systems for land resources assessment*. Oxford University Press, New York, 1987.

[9] W. Burton. Representation of many-sided polygons and polygonal lines for rapid processing. *Communications of the ACM*, pages 166 – 171, March 1977.

[10] G. Bylinsky. Managing with electronic maps. *Fortune Magazine*, pages 237 – 245, April 1989.

[11] T. Sellis C. Faloutos and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proc. of the SIGMOD conference*, pages 426 – 439, 1987.

[12] N. S. Chang. *Image analysis and image database management.* UMI Research Press, Ann Arbor, Michigan, 1981.

[13] S. K. Chang and L. K. Kunii. Pictorial database systems. *IEEE Computer*, 14(11), Nov. 1981.

[14] C. H. Chien and T. Kanade. Distributed quadtree processing. In *Proceedings of Symposium on the Design and Implementation of Large Spatial Databases*, 1989.

[15] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Computing*, pages 181 – 186, 1976.

[16] C. R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19:335 – 348, 1982.

[17] C. S. Ellis. Concurrent search and insertation in 2-3 trees. *Acta Informatica*, 14(1):63 – 86, 1980.

[18] R. A Finkel and J. L. Bentley. A data structure for retrival on composite keys. *Acta Informatica*, pages 1 – 9, April 1974.

[19] J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Publishing Company, 1984.

[20] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Mathematical Software*, pages 209 – 226, September 1977.

[21] I. Gargantini. An efficient way to represent Quadtrees. *Communications of the ACM*, 25(12):905 – 910, 1982.

[22] I. Gargantini. Translation, rotation, and superposition of linear quadtrees. *Internation Journal of Man Machine Studies*, 18(3):253 – 263, March 1983.

[23] T. Gonzalez and J. Jaja. Computing arithmetic expressions with algebraic identities is hard. In *Proc. 1979 Conference on Information Sciences and Systems*, pages 167 – 173, 1979.

[24] T. Gonzalez and J. Jaja. On the complexity of computing bilinear forms with 0, 1 constants. *Comput. System Science*, 20:77 – 95, March 1980.

[25] T. Gonzalez and J. JaJa. Evaluation of arithmetic expressions with algebraic identities. *SIAM Journal of Computing*, pages 633 – 661, November 1982.

[26] O. Gunther. *Efficient Structures for Geometric Data Management*. Springer-Verlag, 1988.

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

[28] G. M. Hunter and K. Steiglitz. Operations on image using quad trees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145 – 153, Apr. 1979.

[29] M. Interrante. A real time spatial index for a 3 -dimensional airspace database. In *IEEE 1987 National aerospace and electronics conference*, pages 424 – 436, 1987.

[30] F. S. Hills Jr. *Computer Graphics*. Macmillan Publishing Company, 1990.

[31] E. Kawaguchi and T. Endo. On a method of binary picture representation and its application to data compression. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pages 27 – 35, January 1980.

[32] H. P. Kriegel and Y. S. Kwong. Insertion-safeness in balanced trees. *Information Processing Letters*, 16:259 – 264, 1983.

[33] Y. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Trans. on Software Engineering*, SE-8(3):211 – 222, May 1982.

[34] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional search trees and quadtrees. *Acta Informatica*, pages 23 – 29, 1977.

[35] P. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. on Database Systems*, 6(4):650 – 670, Dec. 1980.

[36] K. Levitz and H. Levitz. *Logic and Boolean Algebra*. Barron Educational Series Inc., 1979.

[37] G. Nagy and S. Wagle. Geographic data processing. *ACM Computing Survey*, pages 139 – 181, 1979.

[38] J. Nievergelt, H. Hinterburger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, pages 38 – 71, March 1984.

[39] M. A. Oliver and N. E. Wiseman. Operations on quadtree encoded images. *Computer Journal*, pages 83 – 91, February 1983.

[40] C. Papadimitriou. *The theory of database concurrency control.* Computer Science Press, 1986.

[41] T. Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press Inc., 1982.

[42] J. B. Rosenberg. Geographical data structures compared: a study of data structures supporting region queries. *IEEE Trans. on Computer-Aided Design*, pages 53 – 67, January 1985.

[43] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. *ACM SIGMOD Record*, 14(4):17 – 31, May 1985.

[44] B. Samadi. B-trees in a system with multiple users. *Information Processing Letters*, 5(4):107 – 112, 1976.

[45] H. Samet. Region representation: Quadtrees from boundary codes. *Communications of the ACM*, 23(3):163 – 170, March 1980.

[46] H. Samet. The Quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187 – 260, June 1984.

[47] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

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

[49] H. Samet, A. Rosenfeld, C. A. Shaffer, R. C. Nelson, Y. G. Huang, and K. Fujimura. Application of hierarchical data structures to geographical information systems: phase IV. Technical report, Computer Science Dept. and Center for Automation Research, 1985.

[50] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using Quadtrees. *Pattern Recognition*, 17(6):647 – 656, 1984.

[51] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics, part I: fundamentals. *IEEE Computer Graphics & Applications*, pages 49 – 68, May 1988.

[52] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics, part II: applications. *IEEE Computer Graphics & Applications*, pages 59 – 75, July 1988.

[53] T. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J Assoc. Comput. Mach.*, 17:715 – 728, 1970.

[54] C. A. Shaffer, H. Samet, and R. C. Nelson. Quilt: a geographic information system based on quadtrees. Technical report, University of Maryland., 1987.

[55] M. I. Shamos. Geometric complexity. *Proc. Seventh Annual ACM Symposium on Theory of Computing*, pages 224 – 233, 1975.

[56] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proceedings of the seventeenth annual IEEE Symposium on the foundations of computer science*, pages 208 – 215, 1976.

[57] M. Shneier. Calculations of geometric properties using quadtrees. *Computer graphics and image processing*, 16:296 – 302, July 1981.

[58] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, pages 32 – 42, 1974.

[59] L. V. Hao T. Matsuyama and M. Nagao. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing*, 26:303 – 318, June 1984.

[60] N. Roussopoulos T. Sellis and C. Faloutos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proc. of the 13th VLDB conference*, pages 507 – 518, 1987.

[61] R.B. Tilove. Set membership classification: A unified approach to geometric intersection problems. *IEEE Trans. on Computers*, pages 874 – 883, October 1980.

[62] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics*, 12(2):214, 1977.

[63] Kevin Weiler. Polygon comparison using a graph representation. *SIGGRAPH Computer graphics*, 14:10 – 18, 1980.