CAR-TR-99
CS-TR-1457

DAAK70-81-C-0059
November 1984

APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS: PHASE III

Hanan Samet
Azriel Rosenfeld
Clifford A. Shaffer
Randal C. Nelson
Yuan-Geng Huang

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

ABSTRACT

This document is the report of Phase III of an investigation into the application of hierarchical data structures to geographical information systems. It deals primarily with enhancements and improvements to the information system package described in the Phase II report (University of Maryland TR-1327), an evaluation of design decisions, and the collection of empirical results to indicate the utility of the software and justify the design decisions. Tasks reported on include: Attribute attachment, DMA SLF compatibility, memory management improvements, and database enhancements.

The support of the Engineer Topographic Laboratories under Contract DAAK70-81-C-0059 is gratefully acknowledged.

PREFACE

This report documents the research conducted under Phase III of Contract DAAK70-81-C-0059/P00007. The report was prepared for the U.S. Army Engineer Topographic Laboratories (ETL), Ft. Belvoir, Virginia 22060. The Contracting Officer's Representative was Mr. Joseph A. Rastatter.

This report was prepared by Hanan Samet, Azriel Rosenfeld, Clifford A. Shaffer, Randal C. Nelson, and Yuan-Geng Huang.

SUMMARY

This document is the final report for Phase III of an investigation of the application of hierarchical data structures to geographical information systems, conducted under Department of the Army Contract DAAK70-81-C-0059/P00007. The purposes of this investigation were twofold: (1) to construct a geographic information system based on the quadtree hierarchical data structure, and (2) to gather statistics to allow the evaluation of the usefulness of this approach to geographic information system organization.

To accomplish the above objectives, in Phase I of the project a database was built that contained three maps supplied under the terms of the contract. These maps described the flood plain, elevation contours, and landuse classes of a region in California. The map regions were represented in quadtree form, and algorithms were developed for basic operations on quadtree-represented regions (set-theoretic operations, point-in-region determination, region property computation, and submap generation). The efficiency of these algorithms was studied theoretically and experimentally.

In Phase II of the project, a quadtree based Geographic Information System was partially implemented, allowing manipulation of images storing area, point and line data. This system included a memory management system to allow manipulation of images too large to fit into main memory, a software package to allow users to edit and update images, database management and map manipulation functions, and an English-like query language with which to access the database.

Phase III of this project primarily dealt with enhancements and alterations to this information system package, an evaluation of some of the design decisions, and the collection of empirical results to indicate the utility of the software and to justify the indicated design decisions. Included with this report is a survey of appropriate data structures for future investigation vis-a-vis the current system.

The particular tasks reported on in this document are:

(a) *Attribute attachment.* Complete support for arbitrarily associating attributes to image classes or polygons is now provided.

(b) *DMA Standard Lineal Format compatibility.* Map images include a header capable of maintaining all information described by the DMA Standard Lineal Format.

(c) *Memory management improvements.* The memory management software has been optimized, resulting in significant speedup of all operations reported in Phase II. Design decisions for such parameters as page size and number of pages allocated per operation are analyzed and empirical results reported. A new B-tree page splitting algorithm is analyzed.

(d) *Database enhancements.* Additional functions are reported, in particular, a function which allows the user to specify a region within a given distance of a set of

polygons contained in an image. Improved algorithms for some functions reported in Phases I and II are also given.

TABLE OF CONTENTS

FIGURES

TABLES

1. Introduction

This paper reports on the current status of an ongoing effort to determine the suitability of applying a class of hierarchical data structures known as *quadtrees* [Klin71, Same84b] to the representation of cartographic data. Previous project reports are presented in [Rose82, Rose83].

Section 2 describes new work done on the quadtree memory management system. Section 3 describes new work done on the quadtree database system. Section 4 describes the attribute attachment capabilities developed for the quadtree database system. Section 5 discusses theoretical considerations for some of the data structures used in this project, as well as theoretical and empirical analysis of some potential alternatives. Section 6 surveys hierarchical data structures for representing point data. Section 7 surveys hierarchical data structures for representing curvilinear data. Section 8 presents our conclusions and plans for future work.

2. Enhancements to the quadtree memory management system

The quadtree memory management system, as reported in Phase II of this project, is based on a structure termed the *linear quadtree*. The leaf nodes making up the quadtree of an image are stored in a list. Each leaf is sorted on a 32 bit key corresponding to the lower left corner and the depth (size) of the node in the tree. The binary representation of the lower left x and y coordinates are bit interleaved, thus creating a key which sorts the node list in the same order in which the leaves would have been visited by a preorder traversal of the original tree, as detailed in Phase II. Each leaf also contains a 32 bit value field. This technique allows for a reduction in storage over pointer based quadtrees which require each node to store 4 pointers to the children and a pointer to the father, in addition to the value field. The linear quadtree technique can be easily implemented in conjunction with a disk based memory management system which maintains only a small part of the image in core at one time. In our system, the sorted list of quadtree leaves had originally been stored in a B^+-tree [Come79] with a page size of 512 bytes, capable of holding up to 60 leaves in a page. For complete details, see the Phase II report.

In addition to specific topics described below, some work was devoted to optimizations and small algorithm modifications to improve the efficiency of the memory management system. Some of these were in the form of code tuning and language dependent optimizations, but some of the most effective were alterations to the address manipulation functions. In particular, the functions which convert coordinate pairs to and from node address keys are frequently used; these have now been optimized by extensive use of table lookup functions. The node insert algorithm has also been optimized. Taken together, these changes to the memory management system have resulted in a reduction by approximately 20% of the time necessary to run most database queries. This is reflected in the timing results presented in Section 3.

In the remainder of this section we report on major modifications and extensions to the memory management system (known henceforth as the kernel). Additionally, a new page splitting algorithm for the B-tree was tested, and empirical results used to determine the optimal B-tree page size are presented.

2.1. GRAY nodes

The B-tree based data structure employed by the memory management system represents a quadtree as a linear sequence of leaf nodes. However, in some proposed applications (e.g., line quadtrees [Same84a] and the multi-resolution approximations reported on in Phase I of this project), it is advantageous to associate information with the internal or GRAY nodes of the quadtree. Fortunately, storing these GRAY nodes in our file structure is easily done. The order of the nodes in the node list corresponds to a preorder traversal of the quadtree. Thus the structure can be naturally modified to include GRAY nodes by inserting them as they would occur in such a traversal. This preserves the relative order of the original leaf nodes, and consequently, most of the algorithms used in processing leaf-only trees can be used on the corresponding GRAY-node trees with only minor modification.

The kernel has been adapted to permit processing of GRAY-node trees. The user callable routines involved in the insertion and merging of quadtree nodes were modified, and a new field was created in the quadtree file header specifying the type of tree contained in the file. Currently, a GRAY-node tree can be built by calling the quadtree building program with an appropriate parameter, and can be displayed. No further development of GRAY-node capabilities has yet been undertaken.

2.2. Optimal B-tree page and buffer pool sizes

The kernel manipulates a buffer pool in which a queue of several B-tree pages is maintained. The most recently referenced page is always at the head of this queue. When the queue is full and a page is needed which is not in the buffer pool, the information on the least recently used page (i.e., the page at the end of the queue) is copied out to disk and replaced by that of the new page, which is then placed at the head of the queue. Since such swapping is a relatively expensive operation, the number of pages in the buffer pool would be expected to have an effect on the run-time efficiency of the database system.

The size of a B-tree page is also an important factor in system performance. The UNIX operating system transfers a minimum of a block of 1024 bytes of information with each read/write request, regardless of the amount actually requested. It is therefore desirable to make the page size a multiple of this transfer size. With a larger page size, however, more time is required to search the list of nodes on the page for the requested key. For this operation, a small page size is desirable.

In order to investigate the effects of page and buffer pool size, four versions of the kernel were produced to use page sizes of 256, 512, 1024, and 2048 bytes, respectively. The quadtree builder program (which creates a quadtree file from a digital raster image) was then run with buffer pool sizes of 5, 8, 10, 12, 16, and 20 pages for each of the four page sizes. For each test run, the execution time and number of page swaps was collected. The number of page swaps indicates the amount of time used reading and writing to disk, as opposed to the amount of time devoted to quadtree manipulation functions. The results of these experiments are presented in Tables 2.1 and 2.2.

As can be seen from Table 2.1, the effect of the page size on the algorithm runtime is present, but not particularly dramatic. There is a general tendency for the times to increase both with the smallest and the largest page sizes, but this increase is generally less than 30%. The increase for small page sizes is primarily because more page swaps are being done without a proportional decrease in the time needed for each swap, as the machine transfer size is a block of 1024 bytes. For large page sizes the time needed to search for a node in the page has become significant, even though the search algorithm was optimized for each page size. The effect on execution time of the number of pages in the buffer pool is generally minor and without any obvious regularity, once a certain minimum value is exceeded. This minimum is apparent only in the results for the 256 byte page size. For the larger page sizes it is evidently less than 5 pages. Since the kernel's B-tree manipulating functions require a minimum buffer pool size of 4 pages, the pool size does not have a major effect on run-time efficiency.

As was expected, the total number of page swaps decreases when either the page size or the number of pages in the buffer pool increases. This makes sense, for as each increases, more core memory is available to the memory management system. If the object is to minimize disk transfers, then it appears to be more efficient to utilize extra space to increase the page size rather than to increase the size of the buffer pool. These results seem to imply that there is a limited locality of reference to quadtree nodes (at least during the execution of the builder program), i.e., beyond the nodes in the immediate neighborhood of the last node fetched, there is little correlation between the probability of finding a sought-after node in a particular page, and that page's position in the most-recently-accessed hierarchy. This explains the leveling off of disk swaps with increasing pool size, which is apparent in Table 2.2. The abrupt decrease in disk accesses which reappears for very large pages with large buffer pools is due to the fact that enough core has been allocated to contain most of the pages in the B-tree structure.

We conclude from these results that our original choice of a B-tree page size of 512 bytes containing 60 nodes was acceptable. A page size of 1024 bytes containing 120 nodes is equally acceptable, and has now been adopted so that our page size matches the natural page size of the UNIX operating system.

Number of Pages	Page Size (in bytes) 256	512	1024	2048
5	782	568	569	642
8	690	544	556	626
10	716	562	535	598
12	684	578	533	582
16	702	553	527	609
20	694	556	520	602

Table 2.1. Execution times for building the floodplain map. Times are reported in seconds.

Number of Pages	Page Size (in Bytes) 256	512	1024	2048
5	52397	17745	7191	3940
8	27865	12272	6241	2836
10	27094	12141	5963	1399
12	26677	11988	5393	256
16	25941	11587	2864	0
20	25419	10723	503	0

Table 2.2. Number of page swaps performed while building the floodplain map.

2.3. A new B-tree page splitting algorithm.

The original B-tree based storage system utilized a 3 for 4 page splitting algorithm which considers only the linear structure in the ordered sequence of nodes, and takes no account of the underlying quadtree. A new paging algorithm was proposed which would split pages along boundaries corresponding to quadtree quadrants. It was hoped that by thus introducing some of the structure of the original quadtree into the B-tree storage scheme, a higher locality of reference would be maintained, reducing the amount of time spent searching for nodes contained in pages of the B-tree not currently in core memory. Particularly, execution times for functions linked closely to the structure of the underlying quadtree, such as merging leaf nodes, would be substantially decreased. It was recognized that implementation of the new scheme would increase the size of the quadtree files, but it was hoped that this would be tolerable when weighed against the increased efficiency.

Implementation of the new paging algorithm required replacing the kernel routines for splitting and merging B-tree pages, and a number of minor modifications to the remainder of the kernel. The same paging tests that had been performed on the original kernel were then performed using the new paging algorithm. The floodplain map was chosen for the building test since its construction involves the most merging. The results of these experiments are summarized in Tables 2.3 to 2.5 below.

In terms of improving the performance of the database system, the results from this experiment were generally disappointing. The execution times in the best cases are slightly lower with the new algorithm, though considerably higher for the worst cases. This can be seen by comparing the timings from this section with those from Section 2.2. Where some improvement does result, it is not significant, and is offset by the enormous increase in the size of the file. As can be seen from Table 2.5, the files produced using the new page splitting algorithm were approximately three times the size of those produced using the original algorithm. The explanation for this increase is the fact that quadrants in a quadtree containing any sizable blobs are very irregularly filled. This is, in fact, the very means by which a quadtree achieves data compression. The inevitable consequence, however, is that when a page overflows and is split along quadrant boundaries, the contents are apt to be very unevenly distributed among the four new pages. The magnitude of the problem can be appreciated by observing that in the floodplain map produced using the quadrant based paging algorithm, out of a total of 384 pages, 166 contained only one node. The rest were, on the average, less than half full. As a result of the great size of the file and the accompanying increase in the number of pages containing node data, disk activity is increased rather than reduced. In the best cases, approximately twice as many page swaps are performed while building the floodplain map with the new paging algorithm as with the original, as can be seen by comparing Tables 2.2 and 2.4. The fact that the execution times are generally comparable indicates that some activities are being performed more efficiently; unfortunately, the gain is not nearly great enough to offset the negative effect of the large file size. Tests with other maps show similar behavior for the builder.

Number of Pages	Page Size (in bytes) 256	512	1024	2048
5	911	731	633	570
8	618	509	507	573
10	633	503	501	556
12	609	501	496	550
16	633	512	494	537
20	631	513	471	524

Table 2.3. Total execution time while building the floodplain map using a quadrant based page-splitting algorithm. Times are reported in seconds.

Number of Pages	Page Size (in bytes) 256	512	1024	2048
5	193702	125437	61427	5188
8	51033	24352	11038	4781
10	45094	21886	10177	4085
12	42504	21231	9943	2997
16	40916	20766	8463	1156
20	40012	20470	6725	247

Table 2.4. Number of page swaps while building the floodplain map using a quadrant-based page-splitting algorithm.

Page Size (bytes)	Size of File (in bytes) Original	Quadrant-based
256	91392	297948
512	79360	228864
1024	78848	205824
2048	90112	176128

Table 2.5. Comparison of file sizes for floodplain maps using the original and quadrant-based page-splitting algorithms.

3. Enhancements to the Quadtree Database System

Work done on the quadtree database system during this phase of the project falls into two categories. The first includes new functions and capabilities given to the system. Included here are functions to maintain the information described by the DMA Standard Lineal Format, and a new function to allow a user to isolate the area of an image within a certain distance of a set of polygons; they are described in this section. The greatest enhancement is the ability to attach arbitrary attributes to polygons and classes of a region image. This is described separately in Section 4. The second category, also covered in this section, deals with enhancements to already existing functions through code optimizations and implementation of improved algorithms. A number of these are described.

The quadtree database system as a whole has been restructured into three "modes." Most functions are in the standard or default mode. Querying a help function will describe all functions of this mode. The second mode is named "Map-edit." All functions described in the Phase II report as part of the Quadtree Editor are part of this mode. Additionally, those functions which access the DMA Standard Lineal Format header are part of this mode. A user may not access any of these functions without first indicating that he wishes to edit a map. Upon entering the Map-edit mode, the user may not access functions belonging to the other modes until the editing session is finished. A query to the help function while in Map-edit mode will describe only Map-edit mode functions.

The third mode is named "Table-edit." This mode is used to alter an attribute table, as detailed in Section 4. Once again, these functions are available to the user only when he has indicated that he wishes to edit an attribute table, and when in Table-edit mode, only Table-edit mode functions are available.

During this phase of the project, the English-like query language described in Phase II has been temporarily abandoned. It was decided that more work needs to be done in order to produce a truly effective query language, and that resources would be better devoted to expanding the database system at present. Future plans include a reworking of the query language. For now, the user accesses the database system directly through a series of LISP function calls.

LISP allows for a very flexible query language based on the composition of a set of simple functions. The naive user needs no knowledge of LISP to use the database system; he needs only the names and required parameters of the database functions. For the experienced user, our implementation allows the use of a powerful programming language. Queries can easily be generated using the list processing functions of LISP which calculate properties over several maps. Future users will be able to easily extend the present system by writting "intelligent" queries; as an example, such queries could generate formatted tables of data from a set of maps.

Database functions available to each mode are listed in Tables 3.1 to 3.3. Those functions not described in detail in Sections 3.1.1 and 4 are described in the Phase II report.

Name	Description
area	Compute area of a map
build	Build an area map
classof	Return class of an object
colordepth	Label each quadtree node by its depth
comp	Return complement of map
concom	Connected component labelling
cursor	Return the position of the cursor
describe	Describe object
display	Display map
forget	Make variable name reusable
handw	Compute enclosing rectangle
help	Help the user
intersect	Intersect maps
istree	Enter map into the database
lnbuild	Build a line map
name	Give an object a name
numbernodes	Label each quadtree node uniquely
pbuild	Build a point map
perimeter	Calculate the perimeter of a map
polyat	Polygon at a point
pointat	Position cursor at a point
pointof	Return a point from a polygon descriptor
pt2poly	Unique name for polygon at a point
regionsearch	Return all points within a radius
width	Average width of polygons
subset	Return a subset of the polygons in a map
tableof	Return attribute table of a map
treeof	Return map-name containing a polygon
typeof	Return type of map (area, etc.)
union	Union of maps
valueat	Value at a given position
window	Return window of map
within	Expand polygons of a map
	Table and map edit manipulation functions
buildtable	Build an attribute table
class==	Generate set of classes matching a condition
cptable	Copy attribute table
edit	Edit a map (enter Map-edit mode)
edittable	Edit attribute table (enter Table-edit mode)
istable	Enter attribute table into database

Table 3.1. Database functions in standard mode.

Name	Description
abort	Abort editing session
change	Change value of a polygon
comment	Add comment to map header
describe	Describe object
display	Display map
head_ed	Edit quadtree header
help	Help the user
insert	Insert point or line
quit	Finish editing
remove	Remove point or line
replace	Replace value of all polygons in class
slf_init	Initialize SLF header
slf_alter	Alter SLF header
slf_view	Display SLF header
split	Split polygon into pieces

Table 3.2. Database functions in Map-edit mode.

Name	Description
abort	Abort the editing session
addclass	Add a new class
cpclass	Copy attributes of a class
delclass	Delete a class
describe	Describe object
editclass	Edit a class
help	Help the user
printclass	Print attributes of a class
quit	Finish editing

Table 3.3. Database functions in Table-edit mode.

3.1. New functions

3.1.1. DMA Standard Lineal Format

One design requirement of this project is that map images must preserve the information content of the DMA Standard Lineal Format (SLF) [DMA83]. The official description of the SLF is divided into four "record" types: the Data Set Identifier (DSI) record, the Segment record, the Feature record, and the Text record. In our system, information contained in the Segment Record is stored by the quadtree. Information contained in the Feature Record is maintained by our Attribute Attachment system (described in Section 4). The Text Record corresponds to the comments which a user may store with an image by use of the COMMENT command, a member of the Map-edit mode functions. The DSI record contains a list of standard information fields which the user may wish to access. The header format described in Phase II of the project contained only a tiny fraction of this information, so a new header system has been developed along with a set of commands to manipulate it. The structure of this header is nearly identical to the structure described in [DMA83], with many of the field names remaining the same. Where names were altered, such alterations were necessary to maintain uniqueness of the field names.

Associated with each map image is a file containing that map's SLF header. The name of this header file is stored in a new field added to the quadtree header, which is part of the image file. Whenever the SLF header is to be accessed, the database system first locates this file name. The description of the SLF provides for 637 bytes of storage for fixed length fields. Following this is a section with a variable number of fixed length records (the Registration Point records), followed by a section containing a variable number of variable length records (the Accuracy Subset records). Our SLF header package manages the file space in the most compact manner possible. However, there is no restriction imposed on how much information may be contained in the variable length portions of the header

The names and sizes of the fields stored in the SLF header are listed in Table 3.4. The entire header is treated as a simple character file, and an offset from the beginning of the file is associated with each field name. The number of Registration Point records is stored in the number_of_registration_points field. As these records are added or deleted, the remainder of the SLF header following the requested record is moved forward or back as necessary. Finding the beginning of a given record N of this set is easily done by adding N*51 (since each record contains 51 bytes of information) to the value of the registration_points_address field.

The beginning of the Accuracy Subset Group is stored in the accuracy_subset_address field, allowing for easy access to the beginning of the list of Multiple Accuracy Outlines. Unfortunately, as each record is of variable length, to find the beginning of the record N it is necessary to recalculate the address for each access. This is done by examining the length of each Accuracy Outline record in sequence, and calculating the address for the beginning of the next record. Cumbersome as this system may appear, an interactive user should see quick response to header queries.

The user is provided with three functions with which to manipulate the SLF header.

1. slf_init
FORMAT: (slf_init <map> <file-name>)

This function initializes the SLF header for a map image. The name of the map is specified, as well as the name of the file which is to be used to store the SLF header. Initially, most fields are set to be empty.

2. slf_alter
FORMAT: (slf_alter <map> <attribute-name>)

This function is used for all alterations to the SLF header. Most fields are of fixed size. For these fields, the old value will be printed, then the user will be prompted to type the new value. If a carriage return is typed with no value, no change will be made. If the attribute name given is registration_point_record, then the user will be asked if he wishes to add, delete, or alter a record. If delete or alter, then he will be asked which record. The record will then either be deleted, or the user will be prompted for the field to be altered and the new value, as appropriate. If the user wishes to add a record, then he will be prompted for each field's value. The new record will be added at the end of the list of registration_point_records.

Likewise, if the attribute-name given is registration_point_record, then the user will be prompted for details on adding, deleting, or altering the record.

3. slf_view FORMAT: (slf_view <map> [<attribute_name>])

This function allows the user to view the SLF header of map <map>. If no attribute name is given, then the entire record will be displayed. For most attribute names, the value will simply be displayed. If the attribute queried is registration_point_record or accuracy_subset_record, then the user will be prompted to indicate which record he wishes to view. If the user specifies "all," then all such records will be displayed. Otherwise, the user specifies a particular record, and this record alone is displayed.

Table 3.4. The list of SLF header fields. Size refers to the number of characters stored.

Name	Size
DSIG	4
product_type	5
data_set_ID	20
edition	3
compilation_date	4
maintenance_date	4
DSIG_reserve	40
DSSG	4
security_classification	1
security_release	2
downgrading_declassification_date	6
security_handling	21
SG_reserve	40
DSPG	4
data_type	3
horizontal_units_of_measure	3
horizontal_resolution_units	5
geodetic_datum	3
ellipsoid	3
vertical_units_of_measure	3
vertical_resolution_units	5
vertical_reference_system	4
sounding_datum	4
latitude_of_origin	9
longitude_of_origin	10
x_coordinate_of_origin	10
y_coordinate_of_origin	10
z_coordinate_of_origin	10
latitude_of_SW_corner	9
longitude_of_SW_corner	10
latitude_of_NE_corner	9
longitude_of_NE_corner	10
total_number_of_features	6
number_of_point_features	6
number_of_linear_features	6
number_of_area_features	6
total_number_of_segments	6
DSPG_reserve	40

Table 3.4 (continued)

Name	Size
DSMP	4
projection	2
projection_parameter_1	10
projection_parameter_2	10
projection_parameter_3	10
projection_parameter_4	10
scale	9
MPG_reserve	40
DSHG	4
edition_code	3
product_specification	15
specification_date	4
specification_amendment_number	3
producer	8
digitizing_system	10
processing_system	10
grid_system	2
absolute_horizontal_accuracy	4
absolute_vertical_accuracy	4
relative_horizontal_accuracy	4
relative_vertical_accuracy	4
height_accuracy	4
data_generalization	1
north_match/merge_number	1
east_match/merge_number	1
south_match/merge_number	1
west_match/merge_number	1
north_match/merge_date	4
east_match/merge_date	4
south_match/merge_date	4
west_match/merge_date	4
HG_reserve	40
DSVG	4
registration_points_address	5
accuracy_subset_address	5
VFAG_reserve	40

Table 3.4 (continued)

Registration Points Group header and records

Name	Size
DSRG	4
number_of_registration_points	3
DSRG_point_id	6
DSRG_latitude	9
DSRG_longitude	10
DSRG_elevation	8
DSRG_x_coordinate	6
DSRG_y_coordinate	6
DSRG_z_coordinate	6

Accuracy Subset Group header and records

DSAG	4
multiple_accuracy_outline_count	2
DSAG_absolute_horizontal_accuracy	4
DSAG_absolute_vertical_accuracy	4
DSAG_relational_horizontal_accuracy	4
DSAG_relational_vertical_accuracy	2
DSAG_number_of_coordinates	2
DSAG_latitude	9
DSAG_longitude	10

3.1.2. The WITHIN function

The WITHIN function allows the user to create a new map from a map containing a collection of polygons. This new map expands the border of all polygons by R units where R is a radius value specified by the user. This function is an important step for answering queries such as "What is the area of all wheat fields within 5 units of the river?" To answer this query, the user would apply the WITHIN function to the map containing only the river polygon (which can be obtained from the landuse map by using the subset function) along with a radius value of 5 units. The resulting map would then be intersected with the wheatfield map, followed by an area query. In LISP the entire query would appear as

(area (intersect wheatfield (within river 5)))

where wheatfield and river are the appropriate maps.

The current algorithm is as follows. Each node of the input map is examined in sequence. WHITE nodes are ignored. Non-WHITE nodes are processed by a helper function which does the actual work of inserting the appropriate nodes into the output tree. This helper function works as follows. First, the square resulting from expanding the current (non-WHITE) node by R pixels in all directions is calculated. Second, the positions of nodes whose size is the same as that of the current node, and which occur within this square, are calculated. They are then inserted into the output tree. The original node is of course inserted during this process. Finally, the edges of the calculated square are filled in by smaller nodes until all pixels of the square have been inserted. The kernel automatically handles the merging of four sibling nodes which have the same value. This algorithm is admittedly inefficient; better algorithms are being designed and will be implemented in the next phase of the project.

Figures 3.1 and 3.2 show the central region of the floodplain and the map of all ACC polygons from the landuse map, along with the extension resulting by calling the WITHIN function with a radius of 8. Table 3.5 shows timing results for the WITHIN function on these two maps extended by radii taking on values from 1 to 8. An Algol-like description of the algorithm follows:

```
procedure WITHIN(INMAP,OUTMAP,R);
/* Create a map OUTMAP which is BLACK at all pixels within R units of a BLACK
   pixel of INMAP. */
begin
 reference map INMAP, OUTMAP;
 value integer R;
 node ND;

 for ND in INMAP do
   if QD_VALUE(ND) ≠ WHITE then
     whelp(ND, OUTMAP, R, SIZE_OF(INMAP));
end;
```

```
/* In the following algorithm, QD_X and QD_Y calculate the X and Y coordinates of
   the node.  QD_XY builds a node from the given X and Y values, and the WIDTH.
   QD_SET sets the value of a node.  */
procedure WHELP(ND,OUTMAP,R,INSIZE)
begin
 reference map OUTMAP;
 value node ND;
 value integer R, INSIZE;
 node NODE1;
 integer TDIST, X, Y, WIDTH, N, W;

 X ← QD_X(ND);
 Y ← QD_Y(ND);
 WIDTH ← WIDTH_OF(ND);
 TDIST ← R;
 if WIDTH ≤ TDIST then
  begin  /* insert nodes of the same size as the original node */
   N ← TDIST/WIDTH;
   W ← (2*N+1)*WIDTH;
   X ← X - N*WIDTH;
   Y ← Y - N*WIDTH;
   TDIST ← TDIST - N*WIDTH;
   for J ← Y step WIDTH until J > Y + 2*N*WIDTH do
     for I ← X step WIDTH until I > X + 2*N*WIDTH do
      if I ≥ 0 and J ≥ 0 and I+WIDTH ≤ INSIZE
               and J+WIDTH ≤ INSIZE then
        QD_INSERT(OUTMAP,QD_SET(QD_XY(NODE1,I,J,WIDTH),BLACK));
 end;
 else    /* original node is the largest node to be inserted */
 begin
  QD_INSERT(QD_SET(QD_XY(NODE1,X,Y,WIDTH),BLACK);
  W ← WIDTH;
 end;
 while TDIST ≠ 0 do
  if TDIST < WIDTH then
    WIDTH ← WIDTH/2;
  else
   begin
    if Y - WIDTH ≥ 0 then
      for I ← X - WIDTH step WIDTH until I > X + W do
        if I ≥ 0 and I + WIDTH < INSIZE then
          QD_INSERT(OUTMAP,QD_SET(QD_XY(NODE1,I,Y-WIDTH,WIDTH),
                                                BLACK));
    if Y+W+WIDTH ≤ INSIZE then
      for I ← X - WIDTH step WIDTH until I > X + W do
        if I ≥ 0 and I + WIDTH < INSIZE then
          QD_INSERT(OUTMAP,QD_SET(QD_XY(NODE1,I,Y+W,WIDTH),BLACK));
    if X - WIDTH ≥ 0 then
      for I ← Y - WIDTH step WIDTH until I > Y + W do
```

```
        if I ≥ 0 and I + WIDTH < INSIZE then
          QD_INSERT(OUTMAP,QD_SET(QD_XY(NODE1,X-WIDTH,I,WIDTH),
                                  BLACK));
      if X+W+WIDTH ≤ INSIZE then
        for I ← Y - WIDTH step WIDTH until I > Y + W do
          if I ≥ 0 and I + WIDTH < INSIZE then
            QD_INSERT(OUTMAP,QD_SET(QD_XY(NODE1,X+W,I,WIDTH),BLACK));
      TDIST ← TDIST - WIDTH;
      X ← X - WIDTH;
      Y ← Y - WIDTH;
      W ← W + 2 * WIDTH;
      WIDTH ← WIDTH / 2;
    end;
end;
```

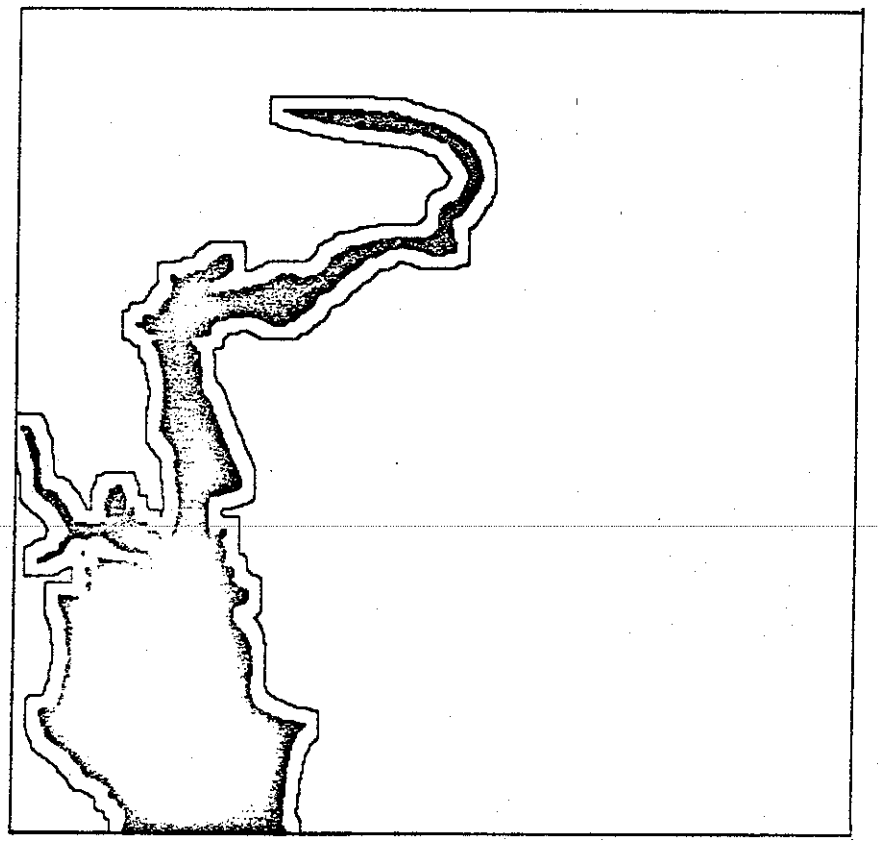


Figure 3.1. The central region of the floodplain with an 8 pixel extension.

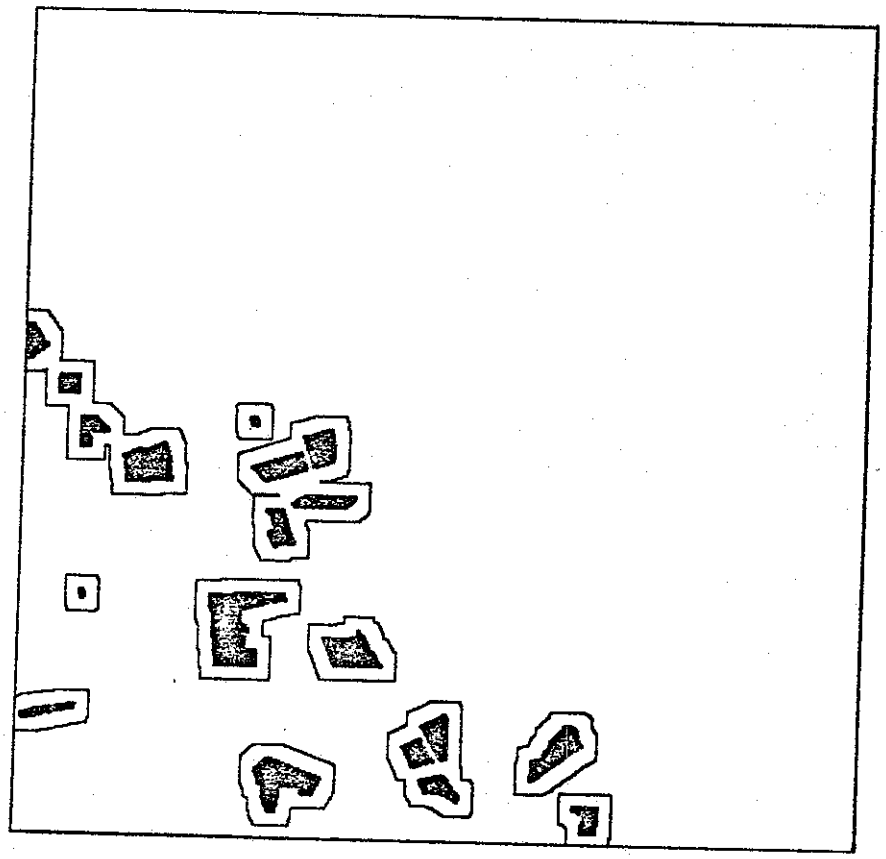


Figure 3.2. The ACC class polygons with an 8 pixel extension.

Distance	Flood time	ACC time
1	1:02.5	1:13.0
2	0:54.5	1:00.9
3	1:53.2	2:17.0
4	1:34.6	1:51.2
5	3:05.2	3:37.3
6	3:05.3	3:37.2
7	4:45.1	5:32.7
8	4:23.1	5:01.0

Table 3.5. Timing results for the WITHIN function calculated on the floodplain center image and the ACC landuse class image for distances ranging from 1 to 8 pixels. The floodplain center image contains 2235 BLACK nodes and 2452 WHITE nodes. The ACC landuse class image contains 1394 BLACK nodes and 1886 WHITE nodes. Times are in minutes.

3.2. Improved algorithms

A number of functions reported in previous phases of this project have been significantly improved by the development of new algorithms. Such results are described in this section.

3.2.1. Table driven traversal

Many functions visit every node in the tree in either a pre-order traversal of the image or an arbitrary order. As each node is visited, some local operation is computed. In some cases it is necessary to determine the postition of the node. One such function is the display function. Each node must be visited, and the block represented by that node is placed on an output device in the appropriate position.

Given a quadtree node, the lower left corner and size of the node can be calculated from the address. The size is of course directly calculated by the depth value, which is easily accessed as it is contained in the fourth byte of the address field. However, the X and Y coordinates of the node are bit interleaved in order to form the address key used in storing the node in the B-tree. Retreiving these coordinates requires a series of arithmetic operations on the address, with the number of these operations related to the depth of the node in the tree.

This time consuming set of operations can be avoided, since the position of the next block of the quadtree in a preorder traversal can be determined knowing the size and position of the previous block. The algorithm used currently by the database system to determine the position of the next block utilizes a two dimensional table with 12 rows (the maximum depth of a quadtree in our database) and three columns corresponding to an X position, a Y position, and a sontype (i.e. a quadrant). This table maintains information on the position of the current node in the traversal, updating this information as each node is visited.

Through use of the table-driven traversal algorithm, the time spent calculating the coordinate positions of the nodes has been cut by more than half. The algorithm is presented below.

```
/* definition and initialization of TRAVTAB */
integer array TRAVTAB[3,2] = {1,0,-1,1,1,0}


procedure DISPLAY(INMAP);
/* Display INMAP by passing a description of each block to a display function. */
begin
 reference map INMAP;
 node ND;
 global integer array TRAVARR[12,3];
 integer I;
```

```
  for I ← 0 step 1 until I = 12 do
   begin          /* initialize array */
    /* x and y values seen at each depth  */
    TRAVARR[I,0] ← TRAVARR[I,1] = 0;
    /* son most recently seen at each depth  */
    TRAVARR[I,2] ← SW;
   end;
  for ND in INMAP do
    DISP_ALL(ND);
end;



Procedure DISP_ALL(ND)
/* calculate coordinates of ND and pass position to display function  */
begin
 value node ND;
 global integer array TRAVARR[12,3];
 integer DEPTH, WIDTH, X, Y;

 DEPTH ← QD_DEPTH(ND);
 WIDTH ← WIDTH_OF(ND);
 X ← TRAVARR[DEPTH,0];
 Y ← TRAVARR[DEPTH,1];
 /* compute local function  */
 DISPLAY_BLOCK(X,Y,WIDTH);
 /* update table for next node  */
 while TRAVARR[DEPTH,2] = NE do
  begin  /* finished quadrant at this depth */
   /* set coordinate to be lower left corner of current quadrant  */
   X ← X - WIDTH;
   Y ← Y - WIDTH;
   TRAVARR[DEPTH,2] ← SW;
   /* move up one level  */
   DEPTH ← DEPTH - 1;
   WIDTH ← WIDTH * 2;
  end;
 /* move to lower left corner of next quadrant; TRAVTAB simply gives multiplyer for
    shift depending on current quadrant  */
 X ← X + WIDTH * TRAVTAB[TRAVARR[DEPTH,2],0];
 Y ← Y + WIDTH * TRAVTAB[TRAVARR[DEPTH,2],1];
 TRAVARR[DEPTH,2] ← TRAVARR[DEPTH,2] + 1;
 for I ← DEPTH step 1 until I = 12 do
  begin  /* adjust lower levels */
   TRAVARR[I,0] ← X;
   TRAVARR[I,1] ← Y;
  end;
end;
```

3.2.2. Quadtree traversal and neighbor finding

Many algorithms for standard quadtree operations can be expressed as simple tree traversals where at each leaf a computation is performed involving that leaf and some or all of its neighbors. Finding the neighbor of a node is somewhat costly as it involves searching the node list. For some operations, it is possible to maintain tables which contain the value, size, and position of those neighboring nodes which have been seen previously during the traversal, as detailed in [Same83b]. To find the value of such a neighbor, it is necessary only to look in the table. The perimeter of an image and connected component labeling can both be computed in this way.

The algorithm makes use of four tables, each the same size as the width of the image. At any instant, the state of the traversal (i.e., after processing m leaf nodes) can be visualized as a staircase (termed an *active border*). The tables maintain information on the value (BLACK or WHITE) of pixels along this active border. An algorithm for computing the perimeter of an image with no explicit neighbor searching is presented below. Note that where this algorithm calls for a computation of the X and Y coordinates, the algorithm of the above section could be used instead.

```
/* New perimeter algorithm; x and y coordinates are computed.  QD_TRAVEL executes
   a helper function over every node in INTREE.  */


integer procedure PERIMETER(INTREE);
begin
 reference map INTREE;
 global integer array XEDGE_WID[SIZE_OF(INTREE)];
 global integer array XEDGE_COL[SIZE_OF(INTREE)];
 global integer array YEDGE_WID[SIZE_OF(INTREE)];
 global integer array YEDGE_COL[SIZE_OF(INTREE)];
 node ND;
 integer I;

 for I ← 0 step 1 until I = SIZE_OF(INTREE) do
  begin
   XEDGE_WID[I] ← YEDGE_WID[I] ← 0;
   XEDGE_COL[I] ← YEDGE_COL[I] ← WHITE;
  end;
 XEDGE_WID[0] ← YEDGE_WID[0] ← SIZE_OF(INTREE);
 for ND in INTREE do
   VALUE ← PERIM(ND);
 return(VALUE);
end;
```

```
integer procedure PERIM(ND)
begin
 value node ND;
 integer WIDTH;
 integer DEPTH;
 integer X, Y, C;
 integer TMP;  /* stores the value of the perimeter for current node */

 DEPTH ← QD_DEPTH(ND);
 WIDTH ← WIDTH_OF(ND);
 C ← QD_VALUE(ND);
 X ← QD_X(ND);
 Y ← QD_Y(ND);
 /* do south edge */
 if C ≠ XEDGE_COL[X] then
   /* adjacent nodes have different colors */
   CNT ← CNT + MIN(WIDTH,XEDGE_WID[X]);
 if WIDTH < XEDGE_WID[X] then
  begin /* new leaf smaller than neighbor - update remainder of border */
   XEDGE_WID[X + WIDTH] ← XEDGE_WID[X] - WIDTH;
   XEDGE_COL[X + WIDTH] ← XEDGE_COL[X];
  end;
 else if WIDTH > XEDGE_WID[X] then
  begin /* new leaf larger than neighbor */
   TMP ← X + XEDGE_WID[X];
   while TMP < (X + WIDTH) do
    begin
     if C ≠ XEDGE_COL[TMP] then
       CNT ← CNT + XEDGE_WID[TMP];
     TMP ← TMP + XEDGE_WID[TMP];
    end;
  end;
 /* update active border */
 XEDGE_COL[X] ← C;
 XEDGE_WID[X] ← WIDTH;
 if (X + WIDTH) = SIZE_OF(INTREE) then /* on border of image */
   if C ≠ WHITE then
     CNT ← CNT + WIDTH;
 /* do west edge */
 if C ≠ EDGE_COL[Y] then
   /* adjacent nodes have different colors */
   CNT ← CNT + MIN(WIDTH,YEDGE_WID[y]);
 if WIDTH < YEDGE_WID[Y] then
  begin /* new leaf smaller than neighbor - update remainder of border */
   YEDGE_WID[Y + WIDTH] ← YEDGE_WID[Y] - WIDTH;
   YEDGE_COL[Y + WIDTH] ← YEDGE_COL[Y];
  end;
 else if WIDTH > YEDGE_WID[Y] then
  begin /* new leaf larger than neighbor */
```

```
    TMP ← Y + YEDGE_WID[Y];
    while TMP < (Y + WIDTH) do
     begin
      if C ≠ YEDGE_COL[TMP] then
        CNT ← CNT + YEDGE_WID[TMP];
      TMP ← TMP + YEDGE_WID[TMP];
     end;
   end;
  /* update active border */
  YEDGE_COL[Y] ← C;
  YEDGE_WID[Y] ← WIDTH;
  if (Y + WIDTH) = SIZE_OF(INTREE) then /* on border of image */
    if C ≠ WHITE then
      CNT ← CNT + WIDTH;
  return(TMP);
end;
```

Map	Old Alg.	New Alg.	Number of Leaves Total	Number of Leaves Black	Number Pages	Page Faults Old	Page Faults New
center	563	78	4687	2235	124	162	124
flood	1196	86	5248	4806	102	161	102
acpmap	745	86	5593	2664	147	210	147

Table 3.6 Perimeter timings to compare two algorithms. The "old" algorithm utilizes a neighbor finding function to locate the South and East neighbors of each node during the traversal. The "new" algorithm utilizes tables to maintain information on those neighbors which have been seen during the traversal (i.e., the Southern and Eastern neighbors). "Number of Pages" indicates the number of B-tree pages required to store the tree; "Page Faults" indicates the number of pages read into core during execution of the algorithm. Since the new algorithm is simply a traversal of the map, each page is read once. Times are in seconds.

3.2.3. A new windowing algorithm

A new windowing algorithm has been developed which is significantly faster for large windows than that described in the Phase II report. The two algorithms are similar in that both are top-down algorithms, and both attempt to insert blocks which are as large as possible into the output tree. For clarity, the algorithm is presented here for use with pointer-based quadtrees.

The new algorithm takes advantage of the fact that the output tree is always completely covered by at most four nodes of the input tree which are of the same size as the output tree. First, these (at most) four nodes are found. If the output tree corresponds to exactly one subtree, then this subtree is copied to the output tree and the algorithm terminates. If all four input tree subtrees are leaves of the same color, then the output tree is a leaf node of this color and the algorithm terminates. Otherwise, the output tree receives a GRAY node and the algorithm is called for each of the four sub quadrants of the output tree. However, for each of the subquadrants of the output tree, the four spanning input tree quadrants will be children of the four input subtrees already found, so very little searching is done. The primary advantage of this algorithm over that described in Phase II is that many fewer nodes need be inserted into the output tree. This is true because many times output nodes that would have been merged by the old algorithm are partitioned between the four input nodes spanning the output subquadrant, all of whom have the same color. In this case, only a single node need be inserted.

Timing statistics for comparision between the two algorithms are presented in Table 3.7. The new algorithm is presented below.

```
node procedure WINDOW(ROOT, INWIDTH, WX, WY, OUTWIDTH);
/* Call WIND, passing in the upper left coordinates and the root of the output tree
   along with the four node of same size as the input tree which span it. */
begin
 value node ROOT;
 value integer INWIDTH, WX, WY, OUTWIDTH;
 node OUTROOT;
 node array Q[4];

 Q[0] ← FINDBLOCK(ROOT,WX,WY,INWIDTH,OUTWIDTH);
 Q[1] ← FINDBLOCK(ROOT,0,0,WX+OUTWIDTH-1,WY,INWIDTH,OUTWIDTH);
 Q[2] ← FINDBLOCK(ROOT,WX,WY+OUTWIDTH-1,INWIDTH,OUTWIDTH);
 Q[3] ← FINDBLOCK(ROOT,WX+OUTWIDTH-1,WY+OUTWIDTH-1,
                  INWIDTH,OUTWIDTH);
 return(WIND(Q,WX-(WX mod WIDTH),WY-(WY mod WIDTH),
             WX,WY,WIDTH,NULL,NULL));
end
```

```
procedure WIND(Q, INX, INY, WX, WY, WID, FATHER, FQD);
/* A top down algorithm which returns, as the QD son of node FATHER, the quadtree
   representing the window (WX,WY,OUTWID) taken from the tree rooted at ROOT
   described by (INX, INY, INWID). */
begin
 reference node array Q[];
 reference node FATHER;
 value integer INX, INY, WX, WY, WID;
 value quadrant FQD;
 node array T[4];
 node R;
 quadrant QD, SUBQD;

 if((not GRAY(Q[0])) and (NODETYPE(Q[0]) = NODETYPE(Q[1]) =
                          NODETYPE(Q[2]) = NODETYPE(Q[3])) then
  /* Output tree is a single leaf node. */
  return(CREATENODE(FATHER,FQD,NODTYPE(Q[0])));
 if(Q[0] = Q[1] = Q[2] = Q[3])
  /* The window coincides with a single input subtree - return a copy. */
  return(COPYSUB(Q[0],FATHER,FQD));
 /* Process each quadrant of the window */
 P ← CREATENODE(FATHER, FQD, GRAY);
 for I in {NW,NE,SW,SE} do
 begin
  /* Compute which four children of the spanning input tree nodes span the I quadrant of the window */
  MODWX ← if I in {NE,SE} then WID else 0;
  MODWY ← if I in {SW,SE} then WID else 0;
  QD ← GETQUADRANT(WID,WID,WX mod WID*2,WY mod WID*2);
  SUBQD ← GETQUADRANT(WID/2,WID/2,WX mod WID,
                      WY mod WID);
  T[0] ← SON(Q[QD],SUBQD);
  QD ← GETQUADRANT(WID,WID,WX+WID-1 mod WID*2,WY mod WID*2);
  SUBQD ← GETQUADRANT(WID/2,WID/2,WX+WID-1 mod WID,
                      WY mod WID);
  T[1] ← SON(Q[QD],SUBQD);
  QD ← GETQUADRANT(WID,WID,WX mod WID*2,WY+WID-1 mod WID*2);
  SUBQD ← GETQUADRANT(WID/2,WID/2,WX mod WID, WY+WID-1 mod
  WID);
  T[2] ← SON(Q[QD],SUBQD);
  QD ← GETQUADRANT(WID,WID,WX+WID-1 mod WID*2,
                   WY+WID-1 mod WID*2);
  SUBQD ← GETQUADRANT(WID/2,WID/2,WX+WID-1 mod WID,
                      WY+WID-1 mod WID);
  T[3] ← SON(Q[QD],SUBQD);
  WIND(T,WX+MODWX-(WX+MODWX mod WID/2),
       WY+MODWY-(WY+MODWY mod WID/2),WX+MODWX,WY+MODWY,
       WID/2,P,I);
end;
```

```
  R ← P;
  MERGE(P,2,2);
  return(R);
end;
```

Size/ Shift	Times Old	New
256		
1	161.7	84.5
2	68.5	55.3
4	43.2	41.7
16	36.5	35.8
64	31.4	30.7
128		
1	37.3	18.3
2	14.7	11.2
4	8.9	8.7
16	7.7	8.0
64	10.5	10.6
64		
1	10.3	5.2
2	4.0	3.4
4	2.5	2.6
16	1.6	1.7
64	2.6	2.6
16		
1	0.7	0.4
2	0.3	0.2
4	0.2	0.2
16	0.1	0.1
64	0.1	0.1

Table 3.7. Windowing timings to compare two algorithms. Size indicates the width and height of the square window. Shift indicates the location of the lower left corner of the window with respect to the input image - i.e., a shift of 2 means that the lower left corner of the window is 2 pixels to the right and above the origin of the input image. Note that a shift by a multiple of a node size will result in no splitting of nodes that size or smaller, thus the shift value affects the difficulty of the operation. Times are measured in seconds.

3.2.4. Set query timings

Timing statistics were gathered to evaluate the efficiency of the set query functions. A list of set queries to be used for gathering empirical results was suggested by ETL. These are used in compiling Tables 3.9 to 3.16 below. Table 3.8 below duplicates Table 3 of the Phase II report illustrating the improved efficiency due to kernel optimizations. Table 3.9 presents these queries as they would be given in our system prior to attribute attachment capabilities. Table 3.10 presents empirical results prior to improvements and optimizations performed on the kernel. Table 3.11 presents a slightly altered set of queries which store intermediate results, reducing the need to recalculate some maps many times. This second set of queries would be more typical of a user's interaction with the database. Tables 3.12 to 3.16 repeat the above experiment while illustrating the use of the attribute attachment system; they also reflect improvements to the kernel.

Land Class	Center Map		Houses Map		Road Map	
	Time	Size	Time	Size	Time	Size
acc	5.0	6341	2.6	9	10.5	30
acp	5.2	26886	2.9	25	10.9	94
ar	5.0	1197	2.2	11	5.4	12
are	4.6	152	1.4	0	4.5	0
avf	12.4	23776	5.5	59	17.0	264
avv	11.9	29685	4.7	50	16.5	341
bbr	3.4	432	1.9	0	7.2	0
beq	2.5	229	1.9	0	5.9	0
bes	4.3	147	1.9	0	4.8	0
bt	6.3	3403	1.6	13	5.9	3
fo	5.9	16952	4.6	4	7.1	30
lr	5.1	948	3.9	0	5.3	1
r	5.6	23147	1.9	4	11.7	94
ucb	2.4	249	1.9	0	6.1	14
ucc	5.6	1018	1.9	4	5.7	34
ucr	2.7	1518	1.3	1	5.6	90
ucw	2.5	305	1.5	2	4.4	4
ues	3.7	1628	1.3	1	5.7	33
uil	2.4	422	1.6	0	4.7	13
uis	2.9	1042	1.4	6	5.9	18
uiw	2.4	186	1.5	4	4.2	6
uoc	2.2	288	1.3	0	4.1	14
uog	2.5	1115	1.4	2	5.1	30
uoo	2.5	490	1.4	1	4.3	11
uop	2.4	213	1.5	10	4.3	8
uov	2.4	238	1.5	9	4.0	6
urh	2.6	167	1.3	3	4.3	3
urs	8.5	26752	5.3	577	22.7	1098
uus	2.7	261	1.4	0	4.2	0
uut	3.6	1928	2.0	2	7.2	18
vv	2.4	108	1.4	0	3.8	2
wo	2.9	0	1.4	0	4.2	0
ws	6.0	3409	1.5	9	9.7	11
wwp	2.5	206	2.1	0	3.9	0

Table 3.8. Timings for the intersection task. Intersection of each class from the landuse map with:

1) the center region of the floodplain map (an area map)
2) the house map (a point map)
3) the road map (a line map)

Size is measured in the number of non-WHITE pixels. Time is measured in seconds.

Table 3.9. 25 set queries

1. (intersect (subset land acp) (subset flood inside))
2. (intersect (subset land acp) (subset flood not inside))
3. (intersect (subset land acp) (subset top not level1 level2))
4. (intersect (subset land acp) (subset top level3 level4 level5))
5. (intersect (subset land acp) (subset top not level3 level4 level5))
6. (intersect (subset land acp) (subset top not level5))
7. (union (subset flood inside) (subset land not avv))
8. (subset land not avv acp)
9. (subset land avv acp)
10. (union (subset flood inside)
    (intersect (subset land not avv) (subset top level1)))
11. (subset land ucc ucr ucw uil uis uiw uoc uog uoo uop urh urs uus)
12a. (intersect (subset land r) top)
  b. (intersect (subset land r) (subset top not level1))
  c. (intersect (subset land r) (subset top not level1 level2))
  d. (intersect (subset land r) (subset top not level1 level2 level3))
  e. (intersect (subset land r) (subset top not level1 level2 level3 level4))
  f. (intersect (subset land r) (subset top not level1 level2 level3 level4 level5))
  g. (intersect (subset land r) (subset top level7 level8 level9 level10 level11))
  h. (intersect (subset land r) (subset top level8 level9 level10 level11))
  i. (intersect (subset land r) (subset top level9 level10 level11))
  j. (intersect (subset land r) (subset top level10 level11))
  k. (intersect (subset land r) (subset top level11))
13. (intersect top (subset flood not inside))
14. (union (subset flood inside) (subset top level5))
15. (intersect (subset flood inside) (subset top level1))

Query Number	Area	Elapsed Time		CPU Time	
		Query Time	Cumulative Time	Query Time	Cumulative Time
1	152	1:06	1:06	0:54.5	0:54.5
2	26734	1:16	2:22	1:08.2	2:01.7
3	12311	2:27	4:49	2:09.7	4:12.6
4	9752	1:59	6:48	1:46.4	5:59.0
5	16813	2:22	9:10	2:08.1	8:07.1
6	25386	2:52	12:02	2:36.2	10:43.3
7	157190	3:26	15:28	3:07.4	13:50.7
8	118928	1:32	17:00	1:25.1	15:15.9
9	56571	0:54	17:54	0:49.0	16:04.9
10	50646	4:19	22:13	3:53.2	19:57.4
11	33777	0:48	23:01	0:43.9	20:41.3
12 a	20752	1:26	24:27	1:17.3	21:58.6
b	20727	2:52	27:19	2:37.5	24:36.2
c	18518	2:36	29:55	2:19.2	26:55.5
d	15591	2:25	32:20	1:59.4	28:54.9
e	12211	1:59	34:19	1:46.4	30:41.4
f	9442	1:45	36:04	1:32.2	32:13.6
g	7416	1:32	37:36	1:22.6	33:36.3
h	5532	1:26	39:02	1:14.9	34:51.2
i	3198	1:14	40:16	1:05.8	35:57.1
j	1108	1:07	41:23	0:57.4	36:54.6
k	8	1:02	42:25	0:56.8	37:49.8
13	42701	1:57	44:22	1:44.7	39:34.5
14	37194	1:23	45:45	1:12.2	40:46.7
15	28446	1:08	46:53	0:59.0	41:45.8

Table 3.10. Timings for 25 set function queries before implementation of attribute attachment. Times are in minutes.

Table 3.11. 25 set queries, storing intermediate results

1. (name 'acpmap (subset land acp))
   (name 'inmap (subset flood inside))
   (intersect acpmap inmap)
2. (name 'outside (subset flood not inside))
   (intersect acpmap outside)
3. (intersect acpmap (subset top not level1 level2))
4. (intersect acpmap (subset top level3 level4 level5))
5. (intersect acpmap (subset top not level3 level4 level5))
6. (intersect acpmap (subset top not level5))
7. (name 'notavv (subset land not avv))
   (union inmap notavv)
8. (subset land not avv acp)
9. (subset land avv acp)
10. (union inmap (intersect notavv (subset top level1)))
11. (subset land ucc ucr ucw uil uis uiw uoc uog uoo uop urh urs uus)
12a. (name 'landr (subset land r))
   (intersect landr top)
   b. (intersect landr (subset top not level1))
   c. (intersect landr (subset top not level1 level2))
   d. (intersect landr (subset top not level1 level2 level3))
   e. (intersect landr (subset top not level1 level2 level3 level4))
   f. (intersect landr (subset top not level1 level2 level3 level4 level5))
   g. (intersect landr (subset top level7 level8 level9 level10 level11))
   h. (intersect landr (subset top level8 level9 level10 level11))
   i. (intersect landr (subset top level9 level10 level11))
   j. (intersect landr (subset top level10 level11))
   k. (intersect landr (subset top level11))
13. (intersect top outside)
14. (union inmap (subset top level5))
15. (intersect inmap (subset top level1))

Query Number	Area	Elapsed Time		CPU Time	
		Query Time	Cumulative Time	Query Time	Cumulative Time
1	152	1:09	1:09	0:54.4	0:54.4
2	26734	0:48	1:57	0:36.4	1:30.7
3	12311	2:03	4:00	1:41.6	3:12.3
4	9752	1:33	5:33	1:17.8	4:30.0
5	16813	2:03	7:36	1:39.9	6:09.9
6	25386	2:41	10:17	2:05.1	8:15.1
7	157190	2:29	13:58	2:58.8	11:14.0
8	118928	2:45	15:31	1:29.5	12:43.5
9	56571	1:04	16:35	0:52.0	13:35.2
10	50646	2:41	19:16	2:08.1	15:43.3
11	33777	0:54	20:10	0:42.3	16:26.2
12 a	20752	1:38	21:38	1:20.6	17:46.9
b	20727	2:25	24:13	2:07.9	19:54.9
c	18518	1:57	26:10	1:07.2	21:42.1
d	15591	1:36	27:46	1:27.5	23:09.7
e	12211	1:20	29:06	1:12.9	24:22.6
f	9442	1:13	30:19	1:01.1	25:24.0
g	7416	1:04	31:23	0:50.4	26:14.5
h	5532	0:51	32:14	0:42.2	26:56.7
i	3198	0:38	32:52	0:33.4	27:30.1
j	1108	0:31	33:23	0:25.7	27:55.9
k	8	0:27	33:50	0:22.4	28:18.3
13	142701	1:38	35:28	1:29.6	29:48.0
14	37194	1:06	36:34	0:58.5	30:46.5
15	28446	0:53	37:27	0:46.6	31:33.2

Table 3.12. Timings for 25 set function queries before implementation of attribute attachment. Times are in minutes.

Table 3.13. 25 set queries with attribute attachment

1. (intersect (subset land (=== class acp)) (subset flood (=== class inside)))
2. (intersect (subset land (=== class acp)) (subset flood (not (=== class inside))))
3. (intersect (subset land (=== class acp)) (subset top (>> elev 2)))
4. (intersect (subset land (=== class acp)) (subset top (and (>> elev 2) (<< elev 6))))
5. (intersect (subset land (=== class acp)) (subset top (or (<< elev 3) (>> elev 5))))
6. (intersect (subset land (=== class acp)) (subset top (not (=== elev 5))))
7. (union (subset flood (=== class inside)) (subset land (not (=== class avv))))
8. (subset land (not (or (=== class avv) (=== class acp))))
9. (subset land (or (=== class avv) (=== class acp))))
10. (union (subset flood (=== class inside))
    (intersect (subset land (not (=== class avv))) (subset top (=== elev 1))))
11. (subset land (=== class u*))

12a. (intersect (subset land (=== class r)) top)
 b. (intersect (subset land (=== class r)) (subset top (>> elev 1)))
 c. (intersect (subset land (=== class r)) (subset top (>> elev 2)))
 d. (intersect (subset land (=== class r)) (subset top (>> elev 3)))
 e. (intersect (subset land (=== class r)) (subset top (>> elev 4)))
 f. (intersect (subset land (=== class r)) (subset top (>> elev 5)))
 g. (intersect (subset land (=== class r)) (subset top (>> elev 6)))
 h. (intersect (subset land (=== class r)) (subset top (>> elev 7)))
 i. (intersect (subset land (=== class r)) (subset top (>> elev 8)))
 j. (intersect (subset land (=== class r)) (subset top (>> elev 9)))
 k. (intersect (subset land (=== class r)) (subset top (>> elev 10)))

13. (intersect top (subset flood (not (=== class inside))))
14. (union (subset flood (=== class inside)) (subset top (=== elev 5)))
15. (union (subset flood (=== class inside)) (subset top (=== elev 1)))

Query Number	Area	Elapsed Time		CPU Time	
		Query Time	Cumulative Time	Query Time	Cumulative Time
1	152	0:41	0:41	0:32.2	0:32.2
2	26734	0:50	1:31	0:40.0	1:12.2
3	12311	1:35	3:06	1:17.0	2:29.2
4	9752	1:20	4:26	1:02.0	3:31.4
5	16813	1:32	5:58	1:16.9	4:48.3
6	25386	1:58	7:56	1:33.4	6:21.7
7	157190	2:11	10:07	1:49.7	8:11.4
8	118928	1:01	11:08	0:52.2	9:03.6
9	56571	0:35	11:43	0:28.6	9:32.2
10	50646	2:46	14:29	2:18.5	11:50.7
11	33777	0:35	15:04	0:29.6	12:20.3
12 a	20752	0:56	16:00	0:46.8	13:07.1
b	20727	1:51	17:51	1:32.1	14:39.2
c	18518	1:38	19:29	1:21.6	16:00.8
d	15591	1:25	20:54	1:10.6	17:11.4
e	12211	1:15	22:09	1:01.7	18:13.1
f	9442	1:09	23:18	0:56.1	19:09.2
g	7416	1:02	24:20	0:50.7	19:59.9
h	5532	0:57	25:17	0:45.2	20:45.1
i	3198	0:51	26:08	0:40.1	21:25.2
j	1108	0:47	26.55	0:37.1	22:02.3
k	8	0:43	27:38	0:34.6	22:36.9
13	142701	1:09	28:47	0:58.2	23:35.1
14	37194	0:51	29:38	0:41.4	24:16.5
15	28446	0:45	30:23	0:35.3	24:51.8

Table 3.14. Timings for 25 set function queries with attribute attachment. Times are in minutes.

Table 3.15. 25 set queries with attribute attachment
Store intermediate results

1. (name 'acpmap (subset land (== class acp)))
   (name 'inmap (subset flood (== class inside)))
   (intersect acpmap inmap)
2. (name 'outside (subset flood (not (== class inside))))
   (intersect acpmap outside)
3. (intersect acpmap (subset top (>> elev 2)))
4. (intersect acpmap (subset top (and (>> elev 2) (<< elev 6))))
5. (intersect acpmap (subset top (or (<< elev 3) (>> elev 5))))
6. (intersect acpmap (subset top (not (== elev 5))))
7. (name 'notavv (subset land (not (== class avv))))
   (union inmap notavv)
8. (subset land (not (or (== class avv) (== class acp))))
9. (subset land (or (== class avv) (== class acp))))
10. (union inmap (intersect notavv (subset top (== elev 1))))
11. (subset land (== class u*))
12a. (name landr (subset land (== class r))
   (intersect landr top)
   b. (intersect landr (subset top (>> elev 1)))
   c. (intersect landr (subset top (>> elev 2)))
   d. (intersect landr (subset top (>> elev 3)))
   e. (intersect landr (subset top (>> elev 4)))
   f. (intersect landr (subset top (>> elev 5)))
   g. (intersect landr (subset top (>> elev 6)))
   h. (intersect landr (subset top (>> elev 7)))
   i. (intersect landr (subset top (>> elev 8)))
   j. (intersect landr (subset top (>> elev 9)))
   k. (intersect landr (subset top (>> elev 10)))
13. (intersect top outside)
14. (union inmap (subset top (== elev 5)))
15. (union inmap (subset top (== elev 1)))

Query Number	Area	Elapsed Time		CPU Time	
		Query Time	Cumulative Time	Query Time	Cumulative Time
1	152	0:41	0:41	0:31.9	0:31.9
2	26734	0:26	1:07	0:20.7	0:52.6
3	12311	1:10	2:17	0:56.9	1:49.5
4	9752	0:58	3:15	0:42.6	2:32.1
5	16813	1:10	4:25	0:56.8	3:28.9
6	25386	1:26	5:51	1:12.3	4:41.2
7	157190	2:03	7:54	1:42.8	6:24.0
8	118928	1:03	8:57	0:52.4	7:16.4
9	56571	0:35	9:32	0:28.9	7:45.3
10	50646	1:34	11:06	1:14.8	9:00.1
11	33777	0:34	11:40	0:29.1	9:29.2
12 a	20752	0:57	12:37	0:46.0	10:16.1
b	20727	1:26	14:03	1:12.9	11:29.0
c	18518	1:15	15:12	1:02.6	12:31.6
d	15591	1:02	16:20	0:51.4	13:23.0
e	12211	0:51	17:11	0:42.6	14:05.6
f	9442	0:45	17:56	0:35.9	14:41.5
g	7416	0:38	18:34	0:31.6	15:13.1
h	5532	0:33	19:07	0:25.8	15:38.9
i	3198	0:26	19:33	0:20.9	15:59.8
j	1108	0:22	19:55	0:17.0	16:16.8
k	8	0:18	20:13	0:14.8	16:31.7
13	142701	0:59	21:12	0:50.4	17:22.1
14	37194	0:42	21:54	0:33.6	17:55.7
15	28446	0:35	22:29	0:27.7	18:23.4

Table 3.16. Timings for 25 set function queries with attribute attachment saving intermediate results. Times are in minutes.

4. Attribute Attachment

It is often desirable to associate some information with an object contained in a map. The user may also wish to ask queries about collections of map objects which have some feature or attribute in common. Storing such information, and answering such queries, is done in our system through the use of the attribute attachment functions described below.

Each area map is associated with what will be referred to as an *attribute class table*. This table is actually a separate file which stores information about a collection of *attribute classes*. An attribute class is simply a list of attributes, and their values. Any number of maps can share an attribute class table. This allows a user to create a database of many maps, say topographic maps, where polygons belonging to a particular attribute class in each map of the set are interpreted in the same way.

Each map image, when created, stores the name of its attribute table file. This file name can be changed by the user if desired. As part of the preparation of an image for use in the database system, each quadtree node is given a value indicating to which class it belongs. This value is an offset into the list of attribute classes, i.e., a node storing the value "5" would be a member of the fifth attribute class of the attribute table associated with that map. Information associated with a node or polygon of an image can be altered by either changing the value of the node so that it becomes a member or another class, or by changing the information stored in the table.

The implementation and storage of attribute tables makes use of the symbolic manipulation and list processing abilities available in the LISP programming language. Attribute classes and attributes are simply arbitrary names picked by the user. The value of an attribute is also arbitrary, and may be either a string or a numeric value. Attribute classes may contain as many attributes as desired.

New attribute tables may be created by use of the function BUILDTABLE. This function prompts the user for classes and their attributes. When a map image is entered into the database system through use of the function ISTREE, the associated attribute table is entered as well. Attribute tables which are not associated with any maps currently known to the database system can be entered by use of the function ISTABLE; they may then be accessed by the user. The function TABLEOF returns the name of the attribute table file associated with a map. COPYTABLE copies an attribute table. This may be useful for creating new tables with only a few changes in the attributes classes.

Attribute tables may be edited through use of the EDITTABLE function. This function places the database system in Table-edit mode, allowing access to the following functions:

1. (editclass <class name>)

Edit class <class name> in the table. The user is prompted as to which attributes are to be altered, added, or deleted.

2. (delclass <class name1> ...)

The named class or classes are deleted from the table.

3. (addclass <class name1> ...)

The named class or classes are added to the table. The editclass function would then be used to add attributes.

4. (cpclass [<table name>] <old class name> <new class name>)

Copy a class (possibly from another table). The new class will have all the attributes and values of the old class. This function is useful for creating new classes which are nearly identical to old classes, with only a few attributes changed.

5. (quit)

Exit Table-edit mode, saving all changes made during the editing session.

6. (abort)

Abort the editing session. The table is restored to its pre-edited state.

7. (help [<function name>])

A description of Table-edit mode functions can be obtained.

8. (printclass [<table name>] [<class name> ...])

Print out information from a table. This function may also be used when not in Table-edit mode. If <table name> is specified, then the function operates on <table name>, otherwise the table currently being edited is used (this only applies when in Table-edit mode). If <class name> is specified, then all attributes and their values of the specified classes are printed. If no class is specified, then a list of all classes in the table will be printed.

Once an attribute table has been created, and associated with a map, the CLASS= function may be used to generate a list of classes from the table which meet some condition, or set of conditions. The format is as follows:

(class= <table name> <condition clause>)

<condition clause> ::=
(>> <attribute name> <value>) |
(<< <attribute name> <value>) |
(== <attribute name> <value>) |
(or <condition clause> <condition clause>) |
(and <condition clause> <condition clause>) |
(not <condition clause>)

This command generates a list of all classes in <table name> which match <condition clause>. The function would normally be used in conjunction with other database functions such as SUBSET. <value> can be either a literal or a regular expression as defined by UNIX. The prefixes ">>" and "<<" are used only for numeric comparisons, the prefix "==" can be used either for numeric comparison or to match either a literal string or regular expression pattern against the attribute. The special <attribute name> "class" matches the name of the class. Those classes for which the condition clause is true are returned. The SUBSET function has also been modified to accept a <condition clause>, returning a map containing those classes matching the specified conditions. Examples of using the SUBSET function with attribute attachment are given in Tables 3.12 and 3.14.

5. Data structure considerations

5.1. Alternative methods for linear quadtree encoding

In the original description of the linear area quadtree, Gargantini [Garg82] suggested that further savings in storage requirements are possible by removing the WHITE nodes from the list of encoded leaves. This should yield an average reduction of 50% in the storage requirements. The missing WHITE nodes could be reconstructed, when necessary, by noting the size and position of the BLACK node retrieved when the list is searched for a given key. If the node requested is not in the tree, then it must be a WHITE node. Likewise, the sequence of WHITE nodes which in the full representation would occur between two BLACK nodes can be inferred, due to the limitations on block size and position that are imposed by the quadtree decomposition method.

Lauzon, *et al.*, have pointed out that this scheme is only suitable for binary images [Lauz84]. A multi-colored image with N node values could be expected to reduce the storage requirements by 1/N of the total for a typical image by deleting those nodes of a specified color. As an alternative Lauzon, *et al.*, propose a data structure which they term a *two dimensional runlength encoding* (2DRE). In this scheme, where several nodes appearing in sequence in the list of leaves all contain the same node value (called a *run*), only the first node is retained. The remaining leaves of the run are removed from the list, and the length of the run is stored with the representative leaf. For a truly random image containing N node values, given a node with value C, the next node should have value C 1/N th of the time. Thus, only slightly more than 1/N of the total storage would be saved from a random image. However, most images represented by quadtrees actually exhibit the characteristic of containing many such runs. Empirical results, as shown in Table 5.1, demonstrate a savings of about 50% for typical multicolored images - similar to that experienced by not storing the WHITE nodes of a binary image.

Point and line data images represented by linear quadtrees have characteristics more like those of binary images than multicolored images. Those nodes which contain a point or line segment are similar to the BLACK nodes of the binary image, with the important difference that each of the node values are unique. Those nodes which do not contain a data point are WHITE. Hence, only WHITE nodes can be removed by the 2DRE scheme. For point and line data representations, 2DRE would not be as efficient as simply removing all of the WHITE nodes as proposed by Gargantini.

It is our belief that when an integrated system utilizing all three data types is desired, all leaves of the quadtree should be retained. Assuming that multicolored images will be stored, removing the WHITE nodes will not result in a significant savings in storage. Although it is true that removing WHITE nodes from point and line images will yield some savings, the additional computing time which would be necessary is too great a penalty. 2DRE encoding will save storage for area images, but not as much savings will result from 2DRE encoding of point and line data.

Tree	Nodes	Runs
Flood	5248	2267
Land	28447	13532
Top	24859	11094

Table 5.1. Comparison of number of nodes in linear quadtree file with number of runs (e.g., 2DRE encoding).

5.2. Comparisons of line and area tree storage

The Phase II report describes a line tree implementation, and four sets of data have been encoded in this fashion. Table 5.2 shows the number of nodes required to encode each data item with our line implementation. This is compared to the number of nodes required to store the data item with our region implementation - i.e. with the pixels on the line BLACK (foreground) and all other pixels WHITE (background).

The City Border map is a closed curve; therefore, it can be considered as the border of a polygon or region. If all of the pixels either on the border or within it are labelled BLACK and all other pixels are labelled WHITE, then 1576 nodes are required. This is less than the number of nodes required to encode the image where only the border pixels are BLACK. One would expect this to be true, since when only the border is BLACK, the image is being divided into three regions - outside, the border, and inside. Viewing this image as a polygon, the border and the inside area are merged into one - causing some of the previously separated pixels to merge into a single node.

In either case, the line implementation is more compact than the area implementation for the same image. However, at this time, algorithms for the database functions already provided for the area representation have not been devised for our line implementation. Algorithms already developed for other known line implementations are not as efficient as their area tree counterparts. However, the potential savings in storage, as demonstrated in Table 5.2, does encourage the investigation of new line data structures. A survey of hierarchical data structures for storing line data is presented in Section 7.

Map	Number of nodes	
	line	area
Road	7729	18700
City Border	835	2281
Powerline	226	1399
Railline	301	1900

Table 5.2. Comparison of node requirements to store line data with our line representation vs. the region quadtree.

5.3. Storing more than one point in a PR quadtree node

Our implementation of the PR quadtree for storing point data was designed to be compatible with the kernel implementation and the node definition for area quadtrees, e.g., one long word for the address field, and one long word for the data field. However, this may not be the most efficient implementation. One important consideration in efficiency is the number of points in a page or node. Matsuyama, *et al.* [Mats84] compare PR quadtrees and K-d trees with differing page sizes.

As the number of points allowed in a node increases, the amount of work required to search the page for a particular node also increases. However, the average number of points stored in a page may also increase, i.e., few WHITE nodes will be stored in the file. Additionally, points within a given node will tend to be near each other; thus, there may a stronger correlation between the most recently found node and the next node to be fetched. Hence, the page size may affect the percent of the time which the point being searched for lies in core.

Space needed to store an image is affected by the tradeoff between overhead for the address (1 long word per node regardless of the number of points stored) versus the utilization of the node (i.e., the number of points / the number of nodes in the image).

In future work, we plan to alter the kernel so that a given image file may have nodes of arbitrary size. The node size for an image will be given when the image is built, and each node in the map will be of this size. Once this has been implemented, it will be possible to store more points in a quadtree node if this is desirable.

Time considerations cannot be tested since the kernel as of yet does not allow variable node sizes. However, empirical results can be obtained for storage requirements. Table 5.3 shows the results of storing the points from the house map in a PR quadtree with node sizes ranging from 1 (as it is now) to 15. Tables 5.4 to 5.7 show similar results for the average of 10 random images containing 10, 100, 1000, and 10,000 points, respectively.

Bin Size	Number of nodes	Number pts/node	% of node Utilized	File Size	% of 1 pt/node
1	1906	0.42	42.29	3812	100.0
2	994	0.81	40.54	2982	78.2
3	703	1.15	38.22	2812	73.8
4	559	1.44	36.05	2795	73.3
5	451	1.79	35.74	2706	71.0
6	373	2.16	36.01	2611	68.5
7	316	2.55	36.44	2528	66.3
8	289	2.79	34.86	2601	68.5
9	271	2.97	33.05	2710	71.1
10	244	3.30	33.03	2684	70.4
11	220	3.66	33.31	2640	69.3
12	196	4.11	34.27	2548	66.8
13	187	4.31	33.16	2618	68.7
14	169	4.77	34.07	2535	66.5
15	157	5.13	34.23	2512	65.9

Table 5.3. Statistics for House map with 806 points

Bin Size	Number of nodes	Number pts/node	% of node Utilized	File Size	% of 1 pt/node
1	22.0	0.4545	45.45	44.0	100.0
2	12.7	0.7874	39.37	38.1	86.6
3	6.1	1.6393	54.64	24.4	55.5
4	4.6	2.1739	54.35	23.0	52.3
5	4.0	2.5000	50.00	24.0	54.5
6	4.0	2.5000	41.67	28.0	63.6
7	4.0	2.5000	35.71	32.0	72.7
8	4.0	2.5000	31.25	36.0	81.8
9	4.0	2.5000	27.78	40.0	90.0
10	1.0	10.0000	100.00	11.0	25.0

Table 5.4. Statistics for average of 10 random images - 10 points

Bin Size	Number of nodes	Number pts/node	% of node Utilized	File Size	% of 1 pt/node
1	214.0	0.4673	46.73	428.0	100.0
2	108.1	0.9251	46.25	324.3	75.8
3	69.4	1.4409	48.03	277.6	64.9
4	54.4	1.8382	45.96	272.0	63.6
5	47.5	2.1053	42.11	285.0	66.6
6	38.2	2.6178	43.63	267.4	62.5
7	31.3	3.1949	45.64	250.4	58.5
8	24.4	4.0984	51.23	219.6	51.3
9	20.8	4.8077	53.42	208.0	48.6
10	17.5	5.7143	57.14	192.5	45.0
11	16.6	6.0241	54.76	199.2	46.5
12	16.3	6.1350	51.12	211.9	49.5
13	16.0	6.2500	48.08	224.0	52.3
14	16.0	6.2500	44.64	240.0	56.1
15	16.0	6.2500	41.67	256.0	59.8

Table 5.5. Statistice for average of 10 random images - 100 points

Bin Size	Number of nodes	Number pts/node	% of node Utilized	File Size	% of 1 pt/node
1	2138.5	0.4676	46.76	4277.0	100.0
2	1089.7	0.9177	45.88	3269.1	76.4
3	730.0	1.3699	45.66	2920.0	68.3
4	541.6	1.8464	46.16	2708.0	63.3
5	413.8	2.4166	48.33	2482.8	58.1
6	335.5	2.9806	49.68	2348.5	54.9
7	292.3	3.4211	48.87	2338.4	54.7
8	267.1	3.7439	46.80	2403.9	56.2
9	255.1	3.9200	43.56	2551.0	59.6
10	245.2	4.0783	40.78	2697.2	63.1
11	233.2	4.2882	38.98	2798.4	65.4
12	215.2	4.6468	38.72	2797.6	65.4
13	193.9	5.1573	39.67	2714.6	63.5
14	179.5	5.5710	39.79	2692.5	63.0
15	159.7	6.2617	41.74	2555.2	59.7

Table 5.6. Statistics for average of 10 random images - 1000 points

Bin Size	Number of nodes	Number pts/node	% of node Utilized	File Size	% of 1 pt/node
1	20375.2	0.4908	49.08	40750.4	100.0
2	10617.7	0.9418	47.09	31853.1	78.2
3	6991.3	1.4303	47.68	27965.2	68.6
4	5184.1	1.9290	48.22	25920.5	63.6
5	4307.2	2.3217	46.43	25843.2	63.4
6	3809.2	2.6522	43.75	26664.4	65.4
7	3418.9	2.9249	41.78	27351.2	67.1
8	3038.8	3.2908	41.13	27349.2	67.1
9	2607.7	3.8348	42.61	26077.0	64.0
10	2200.9	4.5426	45.44	24209.9	59.4
11	1848.1	5.4110	49.19	22177.2	54.4
12	1567.9	6.3780	53.15	20382.7	50.0
13	1366.6	7.3174	56.29	19132.4	47.0
14	1225.3	8.1613	58.29	18379.5	45.1
15	1136.8	8.7966	58.64	18188.8	44.6

Table 5.7. Statistics for average of 10 random images - 10,000 points

5.4. Optimal positioning of quadtrees

Given a digitized image with a specified origin, there is a unique region quadtree which represents it. However, as an image is shifted within the plane, the quadtree does change, typically with a corresponding change in the number of nodes. Li, Grosky, and Jain [Li82] give an algorithm for determining the optimal position of the region quadtree for an image. While the optimal position is expensive to compute, it need only be done once for a given image. Conceivably, the savings in storage for a database of maps could be significant. Dyer [Dyer82] has determined the best, average, and worst case costs for storing a 2^m by 2^m square in a 2^n by 2^n plane ($n > m$). His results indicate that the difference in storage could be as much as a factor of 2^m nodes in the theoretical worst case.

Viewed from the perspective of computing functions on quadtrees, we find that storing images in register is greatly desired. The union and intersection algorithms, for example, operate on the assumption that the two trees are in register. Computing union and intersection on unregistered maps is a problem of difficulty comparable to that of windowing (see Section 3.2.3).

In order to determine how much storage optimal positioning is likely to save in a cartographic database system based on quadtrees, empirical results were obtained by determining the optimal position of the three maps in our test data. The results, shown in Table 5.8, indicate that for complicated images (such as expected for most map images), the positioning is likely to have little effect on the storage requirements. The greatest difference between the best and worst case maps appeared with the floodplain image - the map requiring the least amount of storage in any case.

We would therefore recommend that images be stored in such a way as to be in register whenever possible. This would indicate the desirability of some scheme in which images are broken into regular cells which are along some power of 2 pixels distant from the global origin in large databases spanning many maps.

	Number of Leaves			Position		%
Map	Optimal	Worst	Current	X	Y	Case
Floodplain	5197	9811	5248	48	62	88.7
Topography	24859	28093	24859	0	0	13.0
Landuse	28237	33202	28447	6	18	17.5

Table 5.8. Size and position of the optimal quadtree for map images compared to the current (registered) position. Current leaves, optimal leaves, and worst case columns are measured in number of leaf nodes. The last column gives the percentage difference between the worst case position and the optimal position.

5.5. Neighbor finding in pointer-based quadtrees

A natural byproduct of the tree-like nature of the quadtree is that many basic operations can be implemented as tree traversals (e.g., connected component labeling [Same81], etc.). The difference between them lies in the nature of the computation that is performed at the node. Often, these computations involve the examination of nodes that are adjacent to the node being processed. We call such adjacent nodes *neighbors* and the process of locating them is termed *neighbor finding.* Neighbor finding is also crucial to algorithms for converting between representations (e.g., to construct a quadtree from a boundary code [Same80a] and vice versa [Dyer80], etc.).

In [Same82] algorithms are described and analyzed to compute different kinds of neighbors. The analysis of the execution time of the algorithms was done in terms of the average number of nodes (using a particular image generation model) that needed to be traversed in order to locate the desired neighbor. The average execution time of various neighbor finding primitives was analyzed in [Same82] in terms of the number of nodes that must be visited in locating the desired neighbor. The analysis of each function can be decomposed into two stages corresponding to the process of locating the nearest common ancestor, and then locating the desired neighbor. A random image model was used under which each node is assumed to be equally likely to appear at any position and level in the quadtree. Observe that our notion of a random image differs from the conventional one which implies that every pixel has an equal probability of being BLACK or WHITE. Use of the conventional assumption leads to a very low probability of aggregation (i.e., nodes corresponding to blocks of size greater than 1 pixel). Clearly, for such an image the quadtree is the wrong representation (e.g., a checkerboard).

In order to analyze the second stage, we must also model the distribution of neighbor pairs (i.e., the possible configurations of adjacent nodes of varying sizes). There are a number of models to choose from. In this phase we tried a new model and verified that it correlates very closely with empirical results (as shown in Tables 5.9 to 5.11). For example, suppose that we wish to determine the western neighbor of node 59 in Figure 5.1 that corresponds to the smallest block (it may be GRAY) adjacent to its western side that is of size greater than or equal to the block corresponding to 59 (i.e., M in this case). We term this function GSN. In theory, there are three possible neighbors - i.e., one each of size 1 by 1, 2 by 2, and 4 by 4 at node distances of 6, 5, and 4 respectively. To compute the average value of GSN, the model employed in [Same82], termed the *old model,* treats each of these cases individually and as equally probable - i.e., a node in the same position as 59 makes three contributions to the average value. In contrast, the model introduced in [Same84c] and used in this phase, termed the *new model,* only includes the average contribution of these three cases. Although this modification seems trivial, tests on complex images show that its use leads to very close correlation between theory and practice.

Experiments were conducted on five 512 by 512 maps. Three of these correspond to the overlays furnished by ETL; the remaining two maps are actually thresholded texture images (Figures 5.2 and 5.3). They are the pebble texture (D23) and the stone texture (D7) taken from [Brod77]. For each function four different neighbor finding operations were applied at each leaf node and in all four directions. The operations are named GSN, CSN, GCN, and CCN. GCN is defined analogously to GSN (see above)

with the difference that the direction is diagonal rather than vertical or horizontal as is the case for GSN. GCN and CSN correspond to the smallest blocks that are adjacent to a corner or to a given corner along a side of a node, respectively. For example, for Figure 5.1, GSN(J,E) = K, GSN(J,S) = L, CSN(J,E,SE) = 39, GCN(H,NE) = G, GCN(H,SW) = K, and CCN(H,SW = 38. In the tables that follow the maps are arranged in ascending order of complexity where complexity is the number of nodes in the image. Table 5.9 summarizes the observed values for the individual images, the average value over all five images, the value predicted by the new model, and the value predicted by the old model.

Table 5.9. Average cost of neighbor finding operations.

Operation	Observed						New Model	Old Model
	Flood	Topo	Land	Stone	Pebble	Average		
GSN	3.50	3.60	3.59	3.58	3.56	3.57	3.5	5
CSN	3.66	3.75	3.73	3.73	3.71	3.72	3.67	5.05*
GCN	4.47	4.68	4.63	4.64	4.60	4.60	4.5	6
CCN	4.64	4.83	4.79	4.79	4.75	4.76	4.67	6.07

From Table 5.9 we see that values predicted by the old model were between 25 and 43% above the observed values. In contrast, values predicted by the new model are within 4% of the observed values. We can get a more accurate evaluation of the new model by recalling that the neighbor finding process can be decomposed into two stages. The first stage locates the nearest common ancestor. The old and new models do not differ in the analysis of this stage. There are two cases depending on whether we are seeking a neighbor in the side or corner direction. Table 5.10 shows the empirical results. It is interesting to note how close the model correlates with the observed values.

Table 5.10. Average cost of locating the nearest common ancestor.

Type of Neighbor	Observed						Model
	Flood	Topo	Land	Stone	Pebble	Average	
side	2.01	2.00	2.00	2.00	1.99	2.00	2.00
corner	2.69	2.67	2.66	2.66	2.65	2.67	2.67

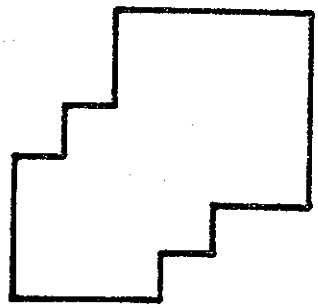
Table 5.11 gives the cost of the second stage of the neighbor finding process. This stage reflects use of our new model and a comparison with the old model reveals the improvement. In particular, we see that for the second stage the values predicted by the old model were between 58 and 100% above the observed values whereas the observed values are within 9% of that predicted by the new model. This is much more reasonable and reinforces use of the new model. Perhaps the most important feature of the new model is the correct prediction that locating neighbors that are of greater than or equal size is cheaper than finding neighbors of equal size. The cost of the latter is simply twice the cost of locating a nearest common ancestor. The reason for the poor

*In [Same82] this value is erroneously computed to be 14/3.

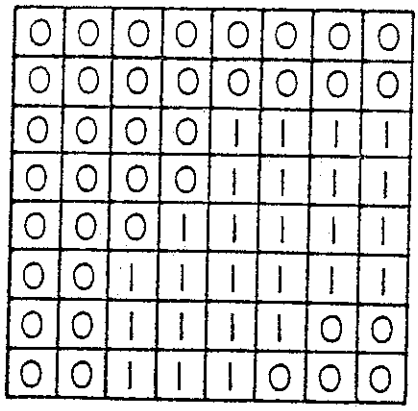
performance of the old model was that all of a node's possible neighbors were individually taken into account when computing the average whereas the new model only included the average contribution of the possible neighbors.

Table 5.11. Average cost of locating the neighbor starting at the nearest common ancestor.

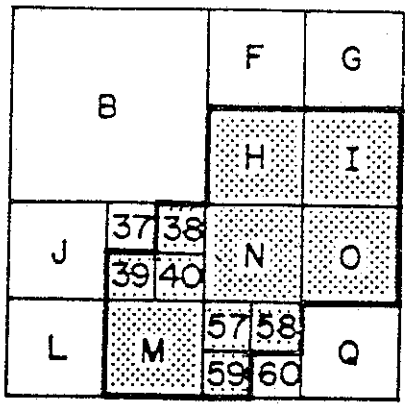
Operation	Observed						New Model	Old Model
	Flood	Topo	Land	Stone	Pebble	Average		
GSN	1.49	1.60	1.59	1.58	1.57	1.57	1.50	3.00
CSN	1.64	1.74	1.73	1.73	1.72	1.71	1.67	3.05
GCN	1.79	2.00	1.97	1.98	1.95	1.94	1.83	3.33
CCN	1.96	2.15	2.13	2.13	2.10	2.09	2.00	3.40



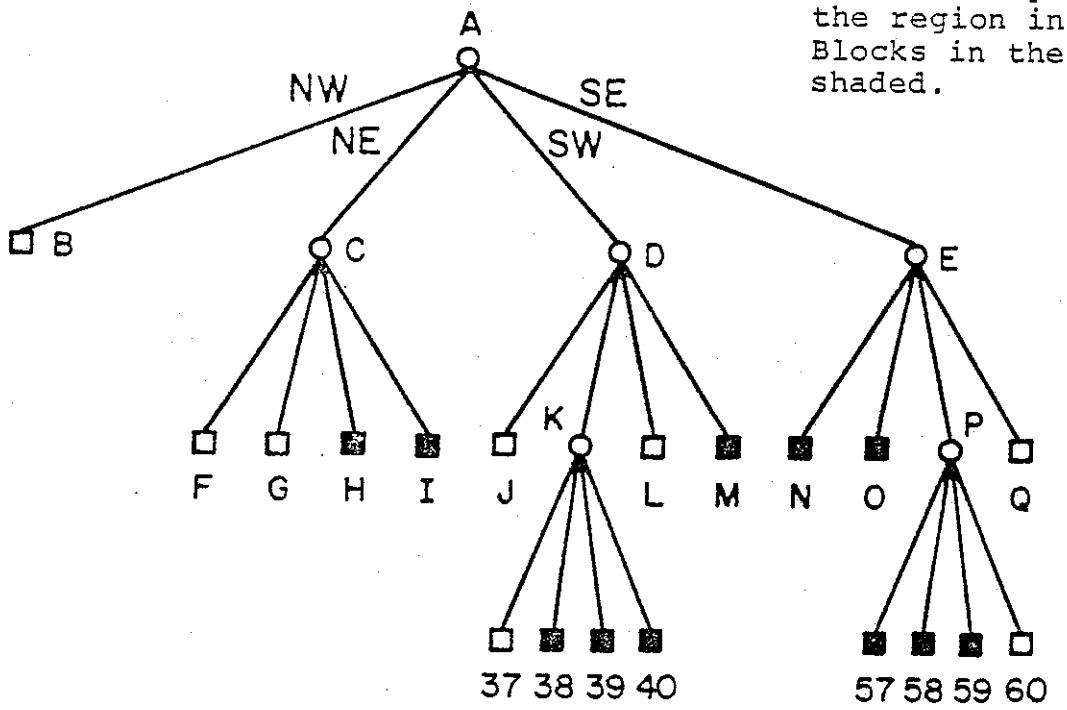
(a) Region



(b) Binary array



(c) Block decomposition of the region in (a). Blocks in the region are shaded.



(d) Quadtree representation of the blocks in (c).

Figure 5.1. Neighbor finding in pointer-based quadtrees. A region, its binary array, its maximal blocks, and the corresponding quadtree.

Figure 5.2. The pebble image.

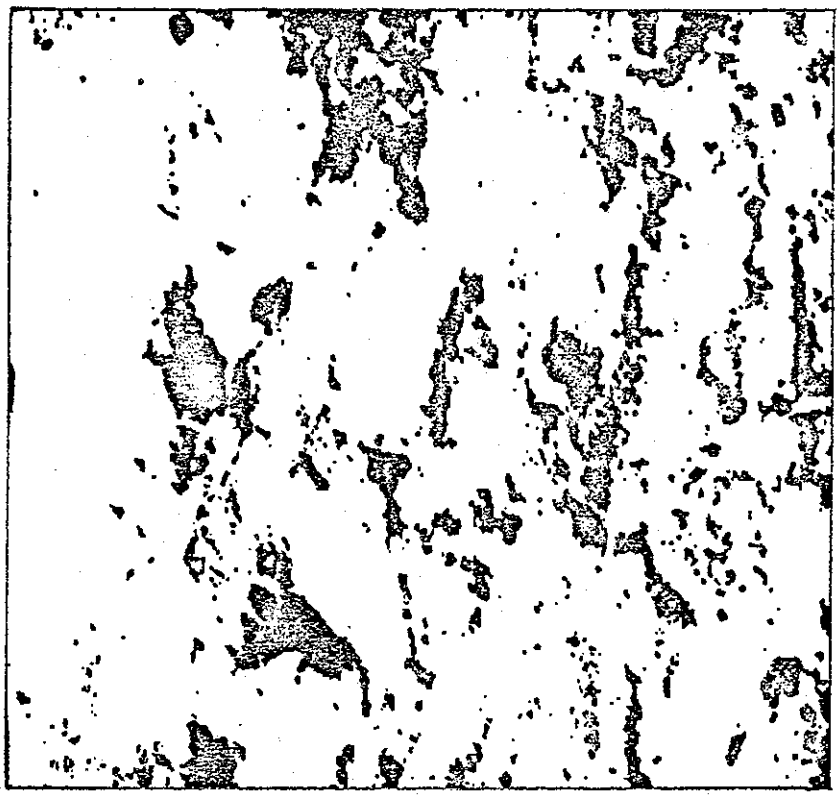


Figure 5.3. The stone image.

6. Point data

This section presents a survey of hierarchical structures for storing point data. The structure used in Phase II of this project is considered in relation with other known structures for solving similar tasks. Structures to be considered for future work under this project will be chosen from those presented in this section.

Multidimensional point data can be represented in a variety of ways. The representation ultimately chosen for a specific task will be heavily influenced by the type of operations to be performed on the data. Our focus is on dynamic files (i.e., the amount of data can grow and shrink at will) and on applications involving search. Knuth [Knut73] lists three typical queries: (1) a *point query* which determines if a given data point is in the data base, and if so, the address corresponding to it; (2) a *range query* (i.e., region search) which asks for a set of data points within a given range (this category includes the partially specified query); (3) a *Boolean query* which consists of the previous type combined with the Boolean operations AND, OR, NOT, etc. A related operation is to find the n nearest neighbors of a given point [Bent75b].

Nievergelt, Hinterberger, and Sevcik [Niev84] group searching techniques into two categories: those that organize the data to be stored and those that organize the embedding space from which the data is drawn. In a more formal sense, the distinction is between *trees* and *tries* respectively. The binary search tree [Knut73] is an example of the former since the boundaries of different regions in the search space are determined by the data being stored. Address computation methods such as radix searching [Knut73] (also known as digital searching) are examples of the latter, since region boundaries are drawn at locations that are fixed regardless of the content of the file. In two dimensions, the distinction between trees and tries can also be seen by comparing the point quadtree [Fink74] with the region quadtree [Klin71]. The former splits the region based on the data while the latter is based on a regular decomposition.

The remainder of this section further elaborates on the point quadtree and the k-d tree [Bent75c]. Next, some representations that are based on the region quadtree (i.e., on a regular decomposition) are discussed and compared with the point quadtree. We conclude with a brief overview of methods that replace the hierarchical structure of quadtrees by address computation. These techniques are aimed, in part, at insuring efficient access to disk data, and are termed ***bucket methods***. In the same context some tree-based methods are also discussed. All of the examples are limited to two dimensions although they can be easily generalized to an arbitrary number of dimensions. It should be borne in mind that our presentation is very brief - i.e., we do not analyze the performance of these methods. Actually, the field of multidimensional data structures is a rapidly developing one, and this discussion is necessarily limited to a detailed presentation of methods that can be viewed as direct applications of a quadtree-like recursive subdivision approach.

6.1. Point quadtrees and k-d trees

The point quadtree [Fink74] is a multi-dimensional generalization of a binary search tree. In two dimensions, each data point is a node in a tree having four sons which are roots of subtrees corresponding to quadrants labeled in order NW, NE, SW, and SE. Each data point is assumed to be unique. The process of inserting into point quadtrees is analogous to that used for binary search trees. In essence, we search for the desired record based on its x and y coordinates. At each node of the tree a four-way comparison operation is performed and the appropriate subtree is chosen for the next test. Reaching the bottom of the tree without finding the record means that it should be inserted at this position. The shape of the resulting tree depends on the order in which records are inserted into it. For example, the tree in Figure 6.1 is the point quadtree for the sequence Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta, and Miami. Deletion of a node is more complex [Same80b].

Point quadtrees are especially attractive in applications that involve search. However, they have also been used to solve a measure problem for rectangular ranges in 3-space [vanL81]. A typical query is one that requests the determination of all records within a specified distance of a given record - i.e., all cities within 50 miles of Washington, DC. The efficiency of the point quadtree lies in its role as a pruning device on the amount of search that is required. Thus many records will not need to be examined. For example, suppose that in the hypothetical data base of Figure 6.1 we wish to find all cities within 8 units of a data point with coordinates (83,10). In such a case, there is no need to search the NW, NE, and SW quadrants of the root (i.e., Chicago with coordinates (35,40)). Thus we can restrict our search to the SE quadrant of the tree rooted at Chicago. Similarly, there is no need to search the NW and SW quadrants of the tree rooted at Mobile (i.e., coordinates (50,10)). Search operations using point quadtrees are analyzed by Bentley and Stanat [Bent75a] and Lee and Wong [Lee77]. Note that the search ranges are usually orthogonally defined regions such as rectangles, boxes, etc. Other shapes are also feasible as the above example demonstrated (i.e., a circle). In order to handle more complex search regions such as polygons, Willard [Will82] defines a *polygon tree* where the $x-y$ plane is subdivided by J lines that need not be orthogonal, although there are other restrictions on these lines. When $J=2$ the result is a point quadtree with non-orthogonal axes.

Our examples of the use of the point quadtree have been limited to two dimensions. The problem with a large number of dimensions is that the branching factor becomes very large (i.e., 2^k for k dimensions) thereby requiring much storage for each node as well as many NIL pointers for terminal nodes. The k-d tree of Bentley [Bent75c] is an improvement on the point quadtree which avoids the large branching factor. In principle, it is a binary search tree with the distinction that at each level of the tree a different coordinate is tested when determining the direction in which a branch is to be made. Therefore, in the two-dimensional case (i.e., a 2-d tree!), we compare x coordinates at the root and at even levels (assuming the root is at level 0) and y coordinates at odd levels. Each node has two sons. Figure 6.2 is the k-d tree corresponding to the point quadtree of Figure 6.1 where the records have been inserted in the same order. Friedman, Bentley, and Finkel [Frie77] report an improvement on the k-d tree which relaxes the requirement of alternating tests at the price of storing at each node an indication of which coordinate is being tested. Using this data structure, termed an *adaptive*

k-d tree, we can construct a balanced k-d tree where records are stored only at the terminal nodes. Figure 6.3 is the adaptive k-d tree corresponding to the point quadtree of Figure 6.1. Prior to constructing such a tree we must know all of the constituent records. Thus its shape is independent of the order in which the records were encountered. However, adding a new record requires rebuilding the tree. Thus it is not a dynamic data structure.

In general, k-d trees are superior to point quadtrees, with one exception: the point quadtree is an inherently parallel data structure and thus the comparison operation can be performed in parallel for the k key values, whereas this cannot be done for the k-d tree. Thus we can characterize the k-d tree as a superior serial data structure and the point quadtree as a superior parallel data structure. Linn [Linn73] discusses the use of point quadtrees in a multiprocessor environment.

6.2. Region-based quadtrees

Although conceivably there are many ways of adapting the region quadtree to represent point data, our discussion is limited to two methods. The first method assumes that the domain of data points is discrete; they are treated as if they are BLACK pixels in a region quadtree. An alternative characterization is to think of the data points as non-zero elements in a square matrix. The resulting data structure is called an *MX quadtree* (MX for matrix) although the term *MX quadtrie* would probably be more appropriate. The MX quadtree is organized in a similar way to the region quadtree. The difference is that leaf nodes are BLACK or empty (i.e., WHITE) corresponding to the presence or absence, respectively, of a data point in the appropriate position in the matrix. For example, Figure 6.4 is the 2^3 by 2^3 MX quadtree corresponding to the data of Figure 6.1. It is obtained by applying the mapping f such that $f(Z)=Z$ div 12.5 to both x and y coordinates. The result of the mapping is reflected in the coordinate values in the Figure.

Each data point in an MX quadtree corresponds to a 1 by 1 square. For ease of notation and operation using modulo and integer division operations, the data point is associated with the lower left corner of the square. This adheres to the general convention followed throughout this presentation that the NE and SE quadrants are closed with respect to the x coordinate and the NW and NE quadrants are closed with respect to the y coordinate. Note that nodes corresponding to data points are not merged whereas this is not the case for empty leaf nodes. For example, the NW and NE sons of node D in Figure 6.4 are NIL and likewise for the NW son of node A. However, it is undesirable to merge nodes corresponding to data points as this results in a loss of the identifying information about the data points. Recall that each data point is different whereas the empty leaf nodes have the absence of information as their common property and thus can be safely merged.

Data points are inserted into an MX quadtree by searching for them. This search is based on the location of the data point in the matrix (e.g., the discretized values of its x and y coordinate in the example of Figure 6.4). An unsuccessful search terminates at a leaf node. If this leaf node is NIL, the space spanned by it may have to be repeatedly subdivided until it is a 1 by 1 square. This process is termed splitting and for a 2^n by 2^n MX quadtree, it will have to be performed at most n times. The shape

of the MX quadtree is independent of the order in which data points are inserted into it. Deletion of nodes is slightly more complex and may require collapsing of nodes - the direct counterpart of the node splitting process outlined above.

The MX quadtree is useful in a number of applications. It serves as a basis of a quadtree matrix manipulation system [Same83c]. It is used by Letelier [Lete83] to represent silhouettes of hand motions to aid in the telephonic transmission of sign language for the hearing impaired. DeCoulon and Johnsen [DeCo76] describe its use in the coding of black and white facsimile for efficient transmission.

The MX quadtree is adequate as long as the domain of the data points is discrete and finite. If this is not the case, then the data points cannot be represented since the minimum separation between the data points is unknown. This leads us to an alternative adaptation of the region quadtree to point data which associates data points (that need not be discrete) with quadrants. We call it a *PR quadtree* (P for point and R for region) although again, the term *PR quadtrie* would probably be more appropriate. The PR quadtree is organized in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., WHITE) or contain a data point (i.e., BLACK) and its coordinates. A quadrant contains at most one data point. For example, Figure 6.5 is the PR quadtree corresponding to the data of Figure 6.1. Orenstein [Oren82] describes an analogous data structure using binary trees rather than quadtrees. Such a data structure could be called a *k-d PR quadtree* or even better simply a *k-d trie.*

Data points are inserted into PR quadtrees in a manner analogous to that used to insert in a point quadtree - i.e., a search is made for them. Actually, the search is for the quadrant in which the data point, say A , belongs (i.e., a leaf node). If the quadtree is already occupied by another data point with different x and y coordinates, say B, then the quadrant must repeatedly be subdivided (termed splitting) until nodes A and B no longer occupy the same quadrant. This may result in many subdivisions, especially if the distance between A and B is very small. The shape of the resulting PR quadtree is independent of the order in which data points are inserted into it. Deletion of nodes is more complex and may require collapsing of nodes - i.e., the direct counterpart of the node splitting process outlined above.

Matsuyama, Hao, and Nagao [Mats84] discuss the use of a PR quadtree in partitioning a point space into "buckets" of a finite capacity. As a bucket overflows, a partition into four, equal-sized squares is made. Anderson [Ande83] makes use of a PR quadtree (termed a *uniform* quadtree) to store endpoints of line segments to be drawn by a plotter. The goal is to minimize pen plotting time by choosing the line segment to be output next whose endpoint is closest to the current pen position. Samet and Webber [Same83c] represent polygonal maps, e.g., Voronoi diagrams, using a variant of the PR quadtree. It has the advantage that edges are represented exactly, thereby avoiding the edge width problem associated with the methods of Hunter and Steiglitz [Hunt79] for polygons.

We use the PR quadtree to store point data in our geographic information system. It is implemented in the form of a variant of the linear quadtree [Garg82]. The PR quadtree is represented as a collection of all the leaf nodes comprising it, where each leaf node is represented by two 32 bit words. The first word contains a pair of numbers

corresponding to the level and a locational code. The latter is a base 4 number corresponding to a sequence of directional codes that locate the leaf along a path from the root of the quadtree. It is analogous to taking the binary representation of the x and y coordinates of a designated pixel in the block (e.g., the one at the lower left corner) and interleaving them (i.e., alternating the bits for each coordinate). For example, let the codes 0, 1, 2, and 3 correspond to quadrants NW, NE, SW, and SE, respectively and assume that Figure 6.4 is a 2^3 by 2^3 image. The block containing Atlanta is represented by the numbers 3 and 60 corresponding to the level (the root is at level 0 in this case) and the locational code respectively. We also have a value field associated with each PR quadtree node which indicates the actual coordinates of the data point. The level and locational code can be combined to form one number that is stored in one word.

6.3. Comparison of point quadtrees and region-based quadtrees

The comparison of the MX, PR, and point quadtrees reduces, in part, to a comparison of their respective decomposition methods. A major difference between the three data structures is in the size of the regions associated with each data point. For the point quadtree there is no *a priori* constraint on the size of the space spanned by the quadtree (i.e., the x and y coordinates of the data points). For both the MX and PR quadtrees the space spanned by the quadtree is constrained to a maximum width and height. All three quadtrees result in the association of one rectangular region with each data point. The point quadtree produces a rectangle which may, at times, be of infinite width and height. For the MX quadtree this region must be a square with a particular size associated with it. This size is fixed at the time the MX quadtree is defined and is the minimum permissible separation between two data points in the domain of the MX quadtree (equivalently, it is the maximum number of elements permitted in each row and column of the corresponding matrix). The PR quadtree also has a square region, and its size depends on what other data points are currently represented by nodes in the quadtree. In the case of the MX quadtree there is a fixed discrete coordinate system associated with the space spanned by the quadtree, whereas no such limitation exists for the PR quadtree. The advantage of such a fixed coordinate system is that there is no need to store coordinate information with a data point's leaf node. The disadvantage is that the discretization of the domain of the data points limits the differentiation between data points.

The size and shape of a quadtree are important from the standpoints of efficiency of both storage and search operations. The size and shape of the point quadtree is extremely sensitive to the order in which data points are inserted into it during the process of building it. This means that for a point quadtree of M records, its maximum depth is $M-1$ (i.e., one record is stored at each level in the tree) while its minimum depth is $\lfloor \log_4(3 \cdot M) \rfloor$ (i.e., each level in the tree is completely full) where we assume that the root of the tree has a depth of 0. In contrast, the shape and size of the MX and PR quadtrees are independent of the insertion order. For the MX quadtree all nodes corresponding to data points appear at the same depth in the quadtree. The depth of the MX quadtree depends on the size of the space spanned by the quadtree and the maximum number of elements permitted in each row and column of the corresponding matrix. For example, for a 2^n by 2^n matrix, all data points will appear as leaf nodes at a depth of n. The size and shape of the PR quadtree depend on the data points

currently in the quadtree. The minimum depth of a PR quadtree for $M>1$ data points is $\lceil \log_4(M-1) \rceil$ (i.e., all the data points are at the same level) while there is no upper bound on the depth in terms of the number of data points. In particular, for a square region of side length s, such that the minimum Euclidean distance separating two points is d, the maximum depth of the quadtree can be as high as $\lceil \log_2((s/d)\cdot\sqrt{2}) \rceil$.

The volume of data also affects the comparison between the three quadtrees. When the volume is very high, the MX quadtree loses some of its advantage since an array representation may be more economical in terms of space, as there is no need for links. While the size of the PR quadtree was seen to be affected by clustering of data points especially when the number of data points is relatively small, this is not a factor in the size of a point quadtree. However, when the volume of data is large and is uniformly distributed, the effect of clustering is lessened and there should not be much difference in storage efficiency between the point and PR quadtrees.

6.4. Bucket methods

All of the data structures discussed above, with the exception of linear quadtree implementations, are primarily designed for in core applications. The problem is that when data is stored in external storage the need to follow pointers may lead to page faults. To overcome this, methods have been designed which collect the points into sets (termed *buckets*) corresponding to the storage unit (i.e., page) of the disk. The remaining task is to organize the access to these buckets; this is often done by replacing the tree structure with an array, thereby facilitating address computation. We term such techniques *bucket methods* and their aim is to insure efficient access to disk data. The simplest bucket method is the fixed grid (or cell) method [Knut73, p. 554; Bent79b] which is popular among cartographers. It divides the space into equal-sized cells (i.e., squares and cubes for two and three-dimensional data respectively) having width equal to the search radius. If data is sought using only a fixed search radius, then the fixed grid is an efficient structure. It is also efficient when points are uniformly distributed (it corresponds to hashing [Knut73]). For a non-uniform distribution it is less efficient, because buckets may be unevenly filled, leading to nearly empty pages as well as long overflow chains. The data structure is essentially a directory in the form of a k-dimensional array with one entry per cell. Each cell may be implemented as a linked list to represent the points within it. Figure 6.6 is an example in which a grid representation for the data of Figure 6.1 is shown for a search radius consisting of a square of size 20 by 20 - i.e., assuming a 100 by 100 coordinate space, we have 25 squares of equal size. Its deficiency is that a fixed size for the blocks which results in both overflow and underflow. The methods presented below are examples of attempts to address this deficiency from both hierarchical and non-hierarchical viewpoints, concluding with a discussion of some related work from the hashing area.

The *Grid File* of Nievergelt, Hinterberger, and Sevcik [Niev84] is a variation of the grid method which relaxes the requirement that cell division lines be equidistant. Its goal is to retrieve records with at most two disk accesses. This is done by using a grid directory consisting of grid blocks which are analogous to the cells of the grid method. All records in one grid block are stored in the same bucket. However, several grid blocks

can share a bucket as long as the union of these grid blocks forms a k-dimensional rectangle (i.e., a convex region) in the space of records. Although the regions of the buckets are piecewise disjoint, together they span the space of records.

The purpose of the grid directory is to maintain a dynamic correspondence between the grid blocks in the record space and the data buckets. The grid directory consists of two parts. The first is a dynamic k-dimensional array which contains one entry for each grid block. The values of the elements are pointers to the relevant data buckets. Usually buckets will have a capacity of 10 to 1000 records. Thus the entry in the grid directory is small in comparison to a bucket. We are not concerned with how records are organized within a bucket (e.g., linked list, tree, etc.). The grid directory may be kept on disk. The second part of the grid directory is a set of k one-dimensional arrays called *linear scales*. These scales define a partition of the domain of each attribute and enable the accessing of the appropriate grid blocks by aiding in the computation of their address based on the value of the relevant attributes. The linear scales are kept in core. It should be noted that the linear scales are useful in guiding a range query by indicating the grid directory elements which overlap the query range.

As an example, consider Figure 6.7 which shows the Grid File representation for the data in Figure 6.1. The bucket capacity is 2 records. There are $k=2$ different attributes. The grid directory consists of 9 grid blocks and 6 buckets labeled A-F. We refer to grid blocks as if they are array elements - i.e., grid block (i,j) is the element in row i (starting at the bottom) and column j (starting at the left) of the grid directory. Grid blocks (2,2), (3,1), and (3,3) are empty; however, they do share buckets with other grid blocks. In particular, grid block (3,1) shares bucket D with grid block (2,1), grid blocks (3,2) and (3,3) share bucket B, while grid blocks (2,2) and (2,3) share bucket E. The sharing is indicated by the broken lines. Figure 6.8 contains the linear scales for the two attributes (i.e., the x and y coordinates). For example, executing a FIND command with $x=80$ and $y=65$ causes the access of the bucket associated with the grid block in row 2 and column 3 of the grid directory of Figure 6.7.

The Grid File is attractive, in part, because of its graceful growth as more and more records are inserted. As the buckets overflow, a splitting process is applied which results in the creation of new buckets and a movement of records. Two types of bucket splits are possible. The first, and most common, is when several grid blocks share a bucket that has overflowed. For example, suppose Boise at (10,80) and Fargo at (15,75) are inserted in sequence in Figure 6.7. Boise is inserted in bucket D because it belongs in grid block (3,1) which currently shares bucket D with grid block (2,1). Fargo also belongs to grid block (3,1); however, bucket D is now full. In this case, we merely need to allocate a new bucket and adjust the mapping between grid blocks and buckets. The second type of a split arises when we must refine a grid partition. It is triggered by an overflowing bucket all of whose records lie in a single grid block (e.g., the overflow of bucket A upon insertion of Kansas City at (30,30) in Figure 6.7). In this case there exists a choice with respect to the dimension (i.e., axis) and the location of the splitting point (i.e., we don't have to split at the midpoint of an interval).

The counterpart of splitting is merging. There are two possible instances where merging is appropriate: (1) *bucket merging*, the most common instance, arises when a pair of neighboring buckets are empty or nearly empty and their coalescing has resulted

in a convex bucket region; (2) *directory merging* arises when two adjacent cross sections in the grid directory each have identical bucket values. For example, in the case of the two-dimensional grid directory of Figure 6.9, where all grid blocks in column 2 are in bucket C and all grid blocks in column 3 are in bucket D, if the merging threshold is satisfied, then buckets C and D can be merged and the linear scales modified to reflect this change. Generally, directory merging is of little practical interest since, even if merging is allowed to occur, it is probable that splitting will soon have to take place.

Merrett and Otoo describe a technique termed *Multipaging* [Merr78, Merr82] which is very similar to the Grid File. It also uses a directory and maintains a set of linear scales called *axial arrays*. In fact, the Grid File uses Multipaging as an index to a paged data structure. A data base that is organized using Multipaging differs from the Grid File in requiring bucket overflow areas. This means that it has a different bucket overflow criterion. Thus it does not guarantee that every record can be retrieved with two disk accesses. In particular, Multipaging makes use of a load factor and a probe factor which are related to the number of overflowing data items. This makes insertion and deletion (as well as bucket splitting and merging) somewhat more complicated than when the Grid File is used.

The *EXCELL* method of Tamminen [Tamm81] is a bintree together with a directory array providing access by address computation. It can also be viewed as an adaptation of extendible hashing [Fagi79] to multidimensional point data. It implements EXHASH, the extendible hashing hash function, by interleaving the most significant bits of the data (analogous to the locational codes discussed in Section 6.2). Similar in spirit to the Grid File, it is based on a regular decomposition and is useful in providing efficient access to, and an efficient representation of, geometric data. It also makes use of a grid directory; however, all grid blocks are of the same size. The principal difference is that grid refinement for the Grid File splits only one interval in two and results in the insertion of a $(k-1)$-dimensional cross section. In contrast, a grid refinement for the EXCELL method splits all intervals in two (thus the partition points are fixed) for the particular dimension and results in doubling the size of the grid directory. Therefore, the grid directory grows more gradually when the Grid File is used, whereas use of EXCELL reduces the need for grid refinement operations at the expense of larger directories in general due to a sensitivity to the distribution of the data. However, a large bucket size reduces the effect of non-uniformity unless the data consists entirely of a few clusters. The fact that all grid blocks define equal sized regions (and convex as well) means that EXCELL does not require a set of linear scales to access the grid directory as is needed for the Grid File.

An example of the EXCELL method is considered in Figure 6.10, which shows the representation for the data in Figure 6.1. Again, the convention is adopted that a rectangle is open with respect to its upper and right boundaries and closed with respect to its lower and left boundaries. The capacity of the bucket is two records. There are $k=2$ different attributes. The grid directory is implemented as an array and in this case it consists of 8 grid blocks (labeled in the same way as for the Grid File) and 6 buckets labeled A-F. Note that grid blocks (2,3) and (2,4) share bucket C while grid blocks (2,1) and (2,2), despite being empty, share bucket D. The sharing is indicated by the broken lines. Furthermore, when a bucket size of 1 is used, the partition of space induced by EXCELL equals that of a PR k-d tree [Oren82].

As a data base represented by the EXCELL method grows, buckets will overflow. This leads to the application of a splitting process which results in the creation of new buckets and a movement of records. As in the case of the Grid file, two types of bucket splits are possible. The first, and most common, is when several grid blocks share a bucket that has overflowed. In this case, a new bucket is allocated and the mapping between grid blocks and buckets is adjusted. The second type of a split arises when a grid partition must be refined; this causes a doubling of the directory. It is triggered by an overflowing bucket that is not shared among several grid blocks (e.g., the overflow of bucket A upon insertion of Kansas City (30,30) in Figure 6.10). The split occurs along the different attributes in a cyclic fashion (i.e., first split along attribute x, then y, then x, etc.). For both types of bucket splits, the situation may arise that none of the elements in the overflowing buckets belongs to the newly created bucket, with the result that the directory will have to be doubled more than once. This is because the splitting points are fixed for EXCELL. For example, this will occur when we attempt to insert Kansas City at (30,30) in Figure 6.10 since the first directory doubling at $y=25$ and $y=75$ will still have Chicago, Omaha, and Kansas City in the same grid block. Thus we see that the size of the EXCELL grid directory is sensitive to the distribution of the data. However, a large bucket size reduces the effect of non-uniformity unless the data consists entirely of a few clusters.

The counterpart of splitting is merging. However, it is considerably more limited in scope for EXCELL than for the Grid File. Also, it is less likely to arise because EXCELL has been designed primarily for use in geometrical applications where deletion of records is not so prevalent. As with the Grid File, however, there are two cases where merging is appropriate, i.e., bucket merging and directory merging.

Both the Grid File and EXCELL organize space into buckets and use directories in the form of arrays to access them. The similarity to the quadtree lies in the mappings induced by the directories (i.e., EXCELL with the region quadtree and the Grid File with the point quadtree). Trees can also be used to access the buckets [Knot71]. Matsuyama, Hao, and Nagao [Mats84] compare a technique of accessing buckets by use of a PR quadtree with one that uses an adaptive k-d tree. Robinson [Robi81] introduces the *k-d-B-tree* which is a generalization of the B-tree to allow multiattribute access. O'Rourke [ORou81, ORou84] makes use of an adaptive k-d tree which he calls a *Dynamically Quantized Space* to access buckets of data for use in multidimensional histogramming to aid in the focusing of the Hough transform. Sloan [Sloa81, ORou84] addresses the same problem as O'Rourke, albeit with a different data structure which he calls a *Dynamically Quantized Pyramid.* It is based on the pyramid data structure [Tani75]. Here the number of buckets is fixed. It differs from the conventional pyramid in that the partition points at the various levels are allowed to vary rather than being fixed and are adjusted as data is entered. The result is somewhat related to a complete point quadtree [Knut75, p. 401] with buckets.

There has also been a considerable amount of work on representing multidimensional point data by use of linear hashing. Linear hashing [Litw80] methods are attractive because they provide for linear growth of the file (i.e., one bucket at a time), without requiring a directory. In contrast, extendible hashing [Fagi79] (e.g., EXCELL) and the Grid File methods require extra storage in the form of directories. When a bucket overflows, the directory doubles in size in the case of extendible hashing, while in

the case of the Grid File it results in the insertion of a (k-1)-dimensional cross-section. Neither EXCELL nor the Grid File need overflow pages, while methods based on linear hashing generally do, although this may be unnecessary. Bit interleaving (e.g., attributed in [Bent75b] to McCreight; also known as a Morton Matrix [Mort66]) is used by a number of researchers [Trop81, Burk83, Oren83, Ouks83, Oren84], to create a linear order on the multidimensional domain of the data. Tropf and Herzog [Trop81] and Orenstein and Merrett [Oren84] discuss its use in range searching. Burkhardt [Burk83], terms it *shuffle* order, and adapts it to linear hashing, in the same way as EXCELL adapts it to extendible hashing, and uses it to evaluate range queries. Ouksel and Scheuermann [Ouks83] call it z order. Orenstein [Oren83] discusses the problems associated with such an approach. He points out that the resulting file may contain a number of sparsely filled buckets which will result in poor performance for sequential access. He goes on to propose a modification which unfortunately, unlike linear hashing, does not result in a bucket retrieval cost of one or two disk reads (for the hash operations). In contrast, directory-based methods such as the Grid File and EXCELL do not suffer from such a problem to the same extent because, since the directory consists of grid blocks and several grid blocks can share a bucket, the sparseness issue can be avoided.

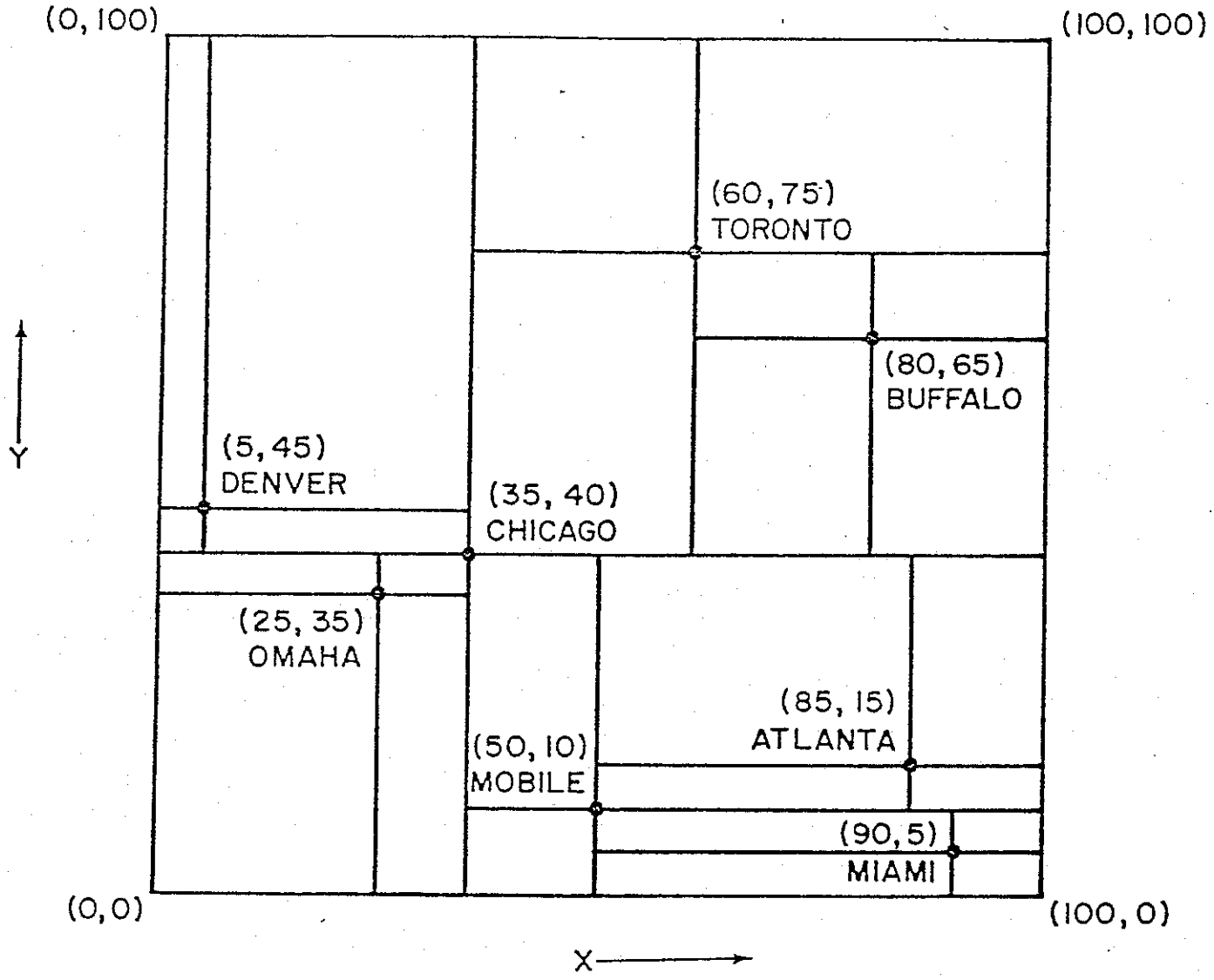


Figure 6.1. A quad tree and the records it represents.

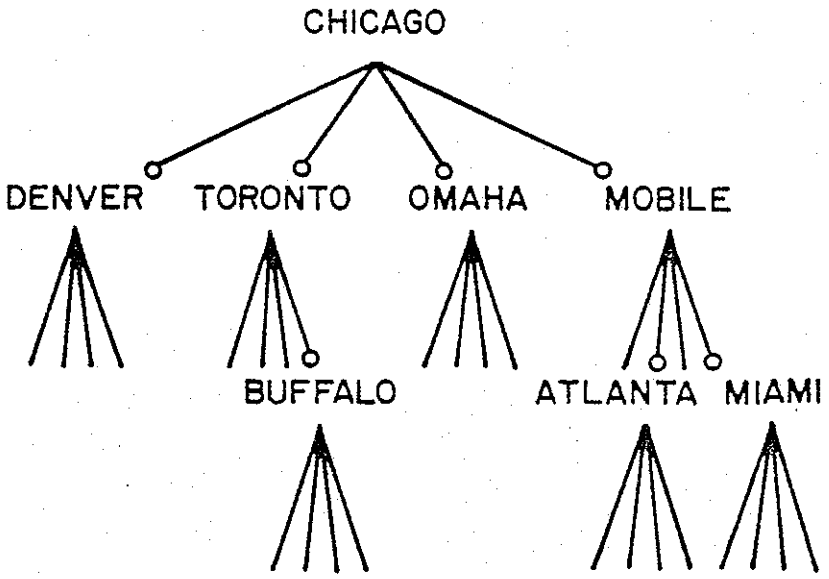


Figure 6.1. (continued)

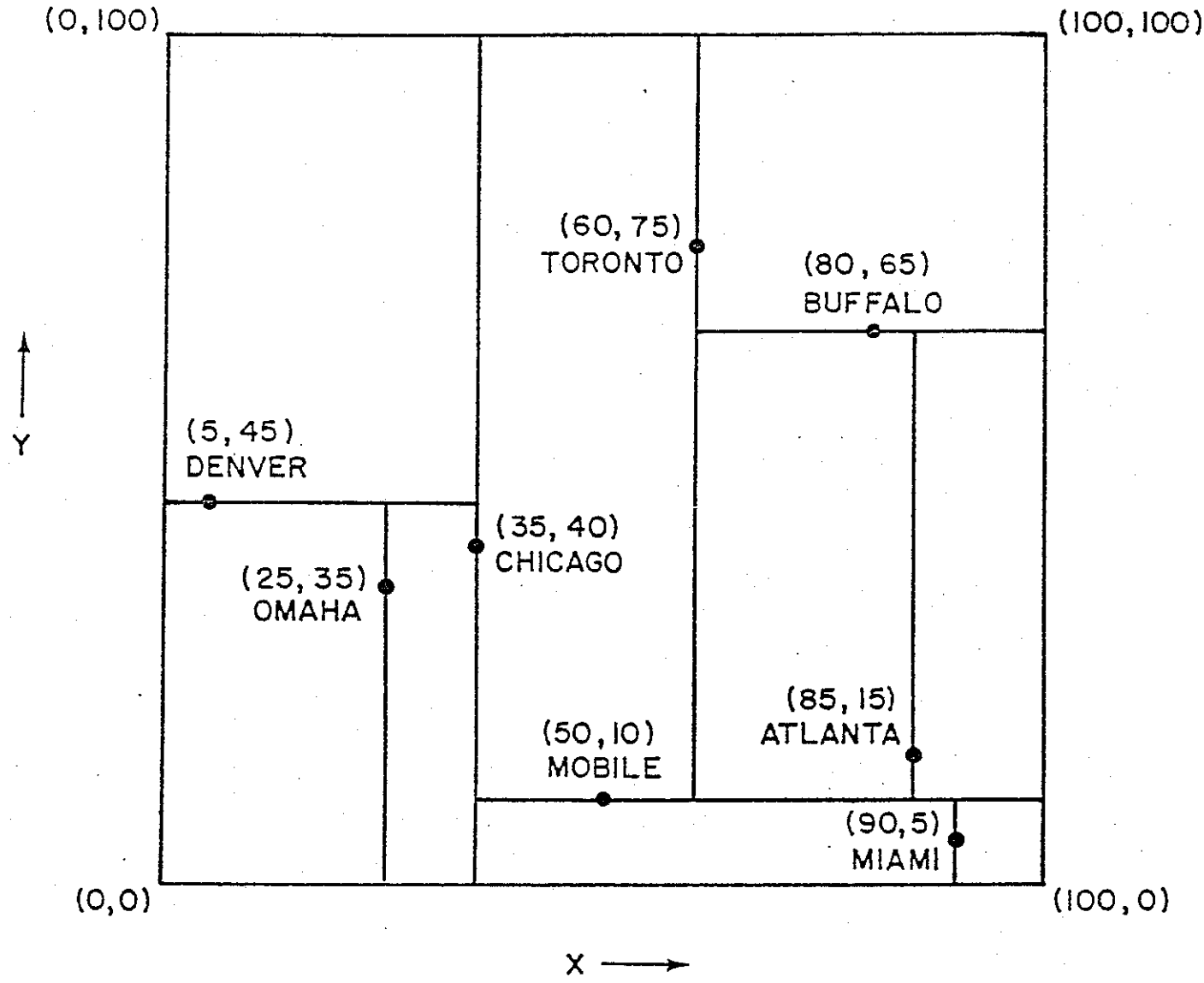


Figure 6.2 A K-d tree and the records it represents.

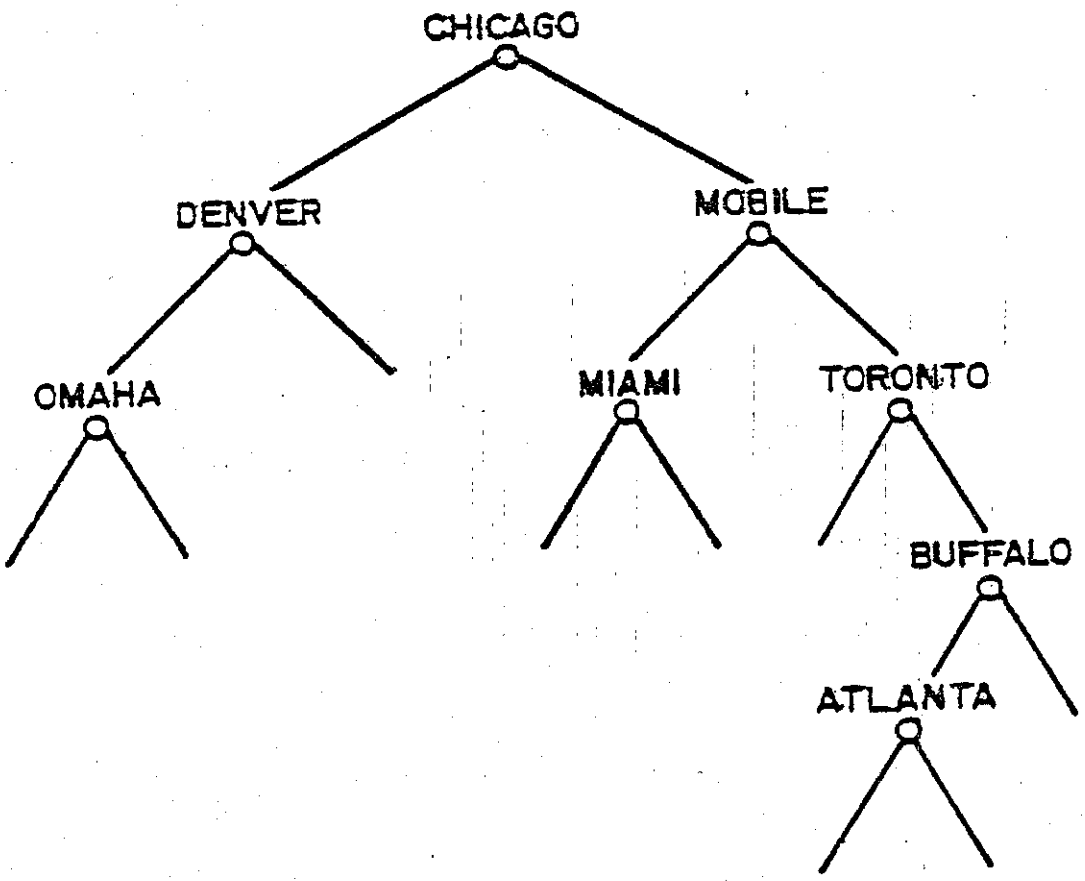


Figure 6.2. (continued)

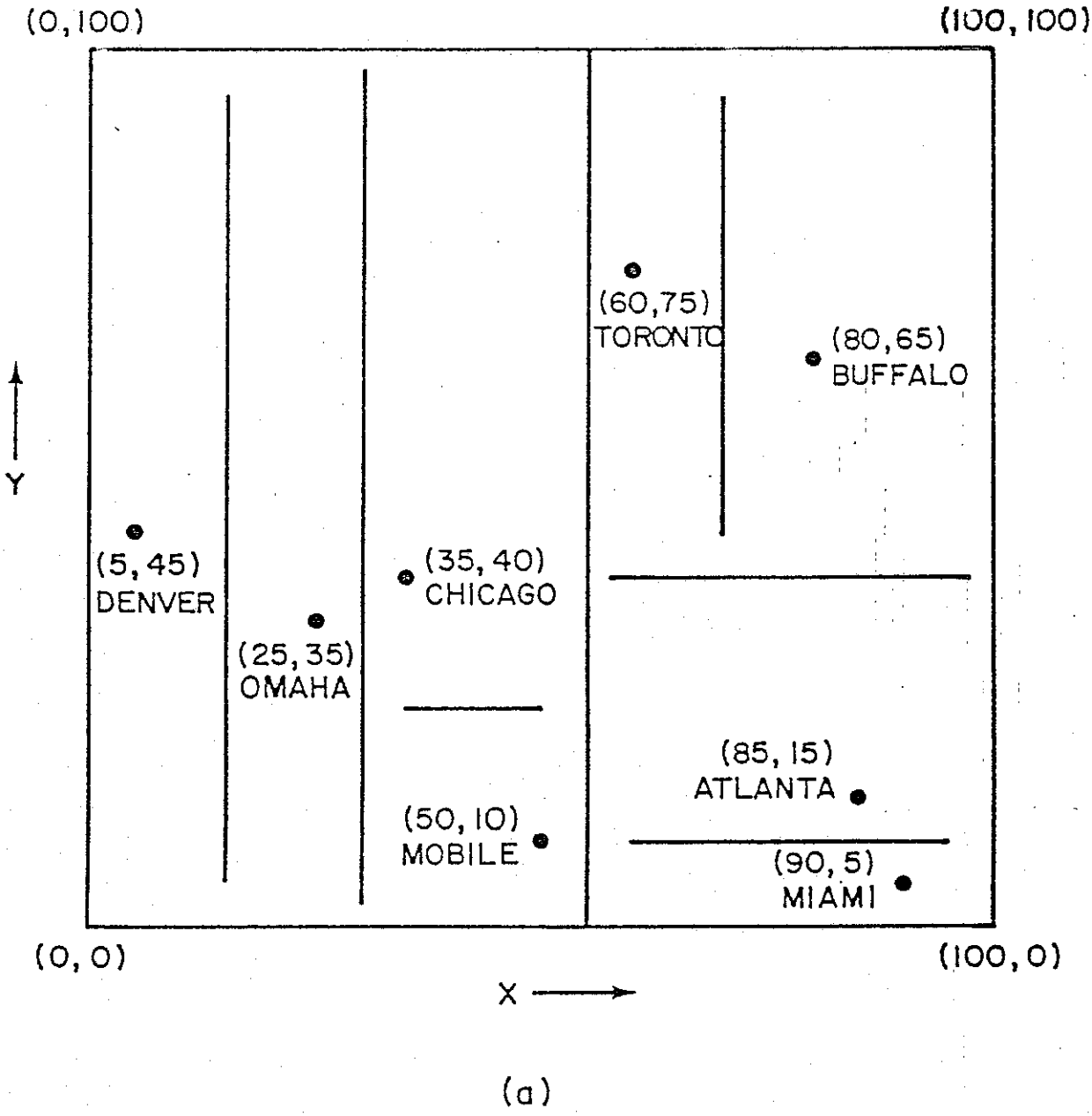
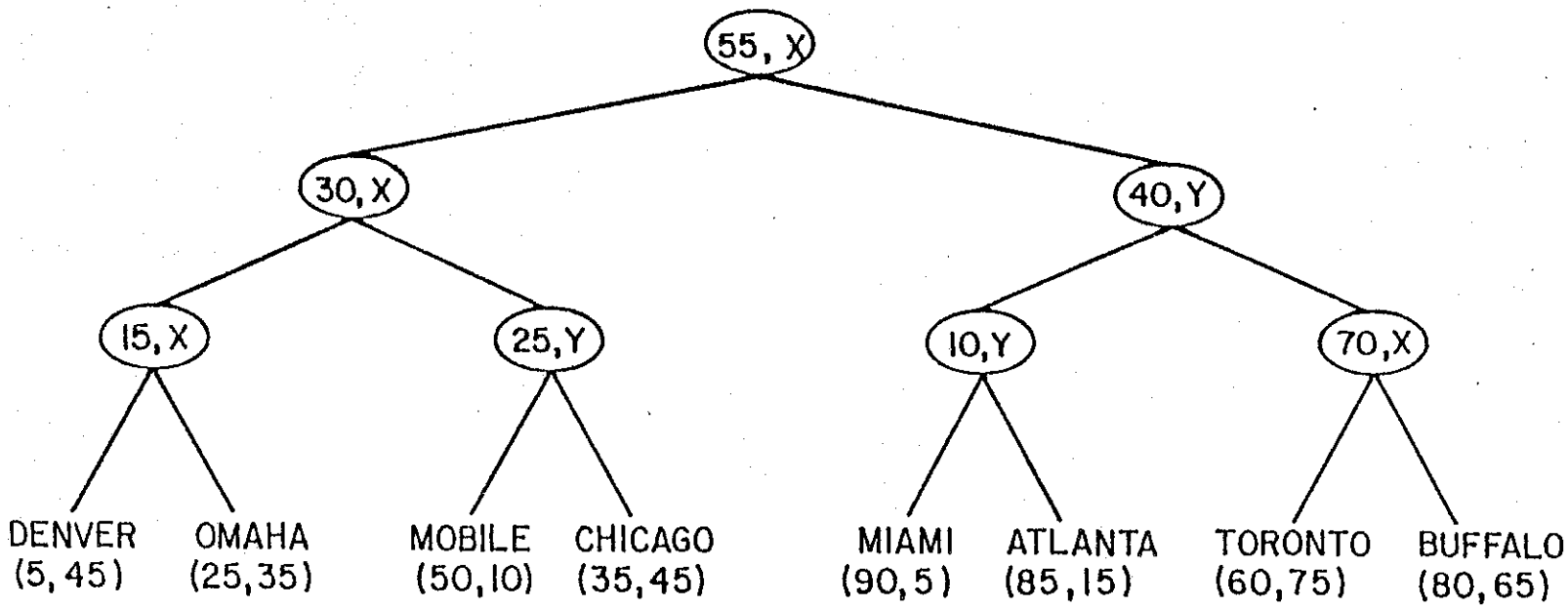


Figure 6.3. Adaptive K-d tree. (a) Set of points in 2-space, (b) 2-d tree.



(b)

Figure 6.3. (continued)

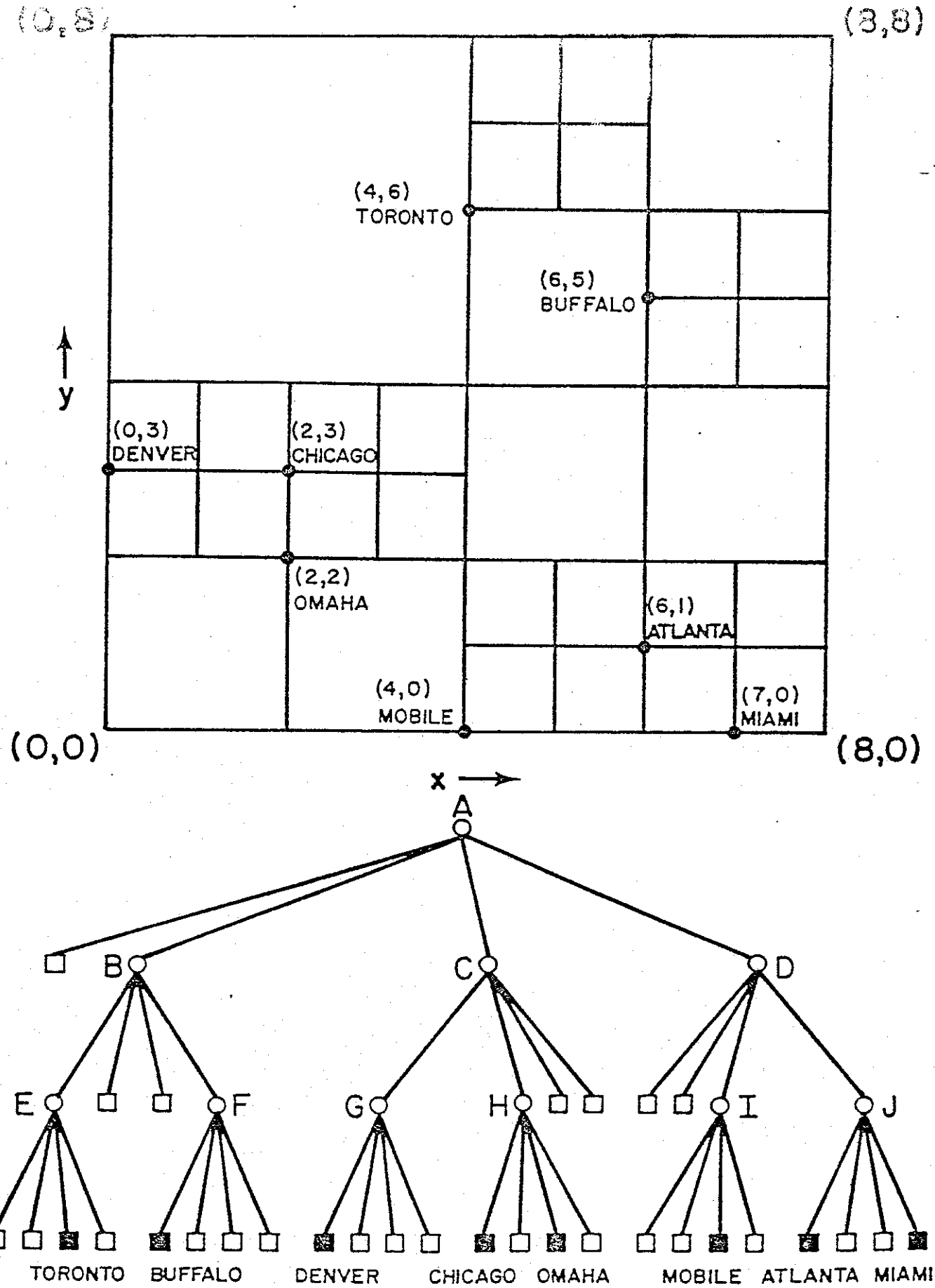


Figure 6.4. An MX quadtree and the records it represents.

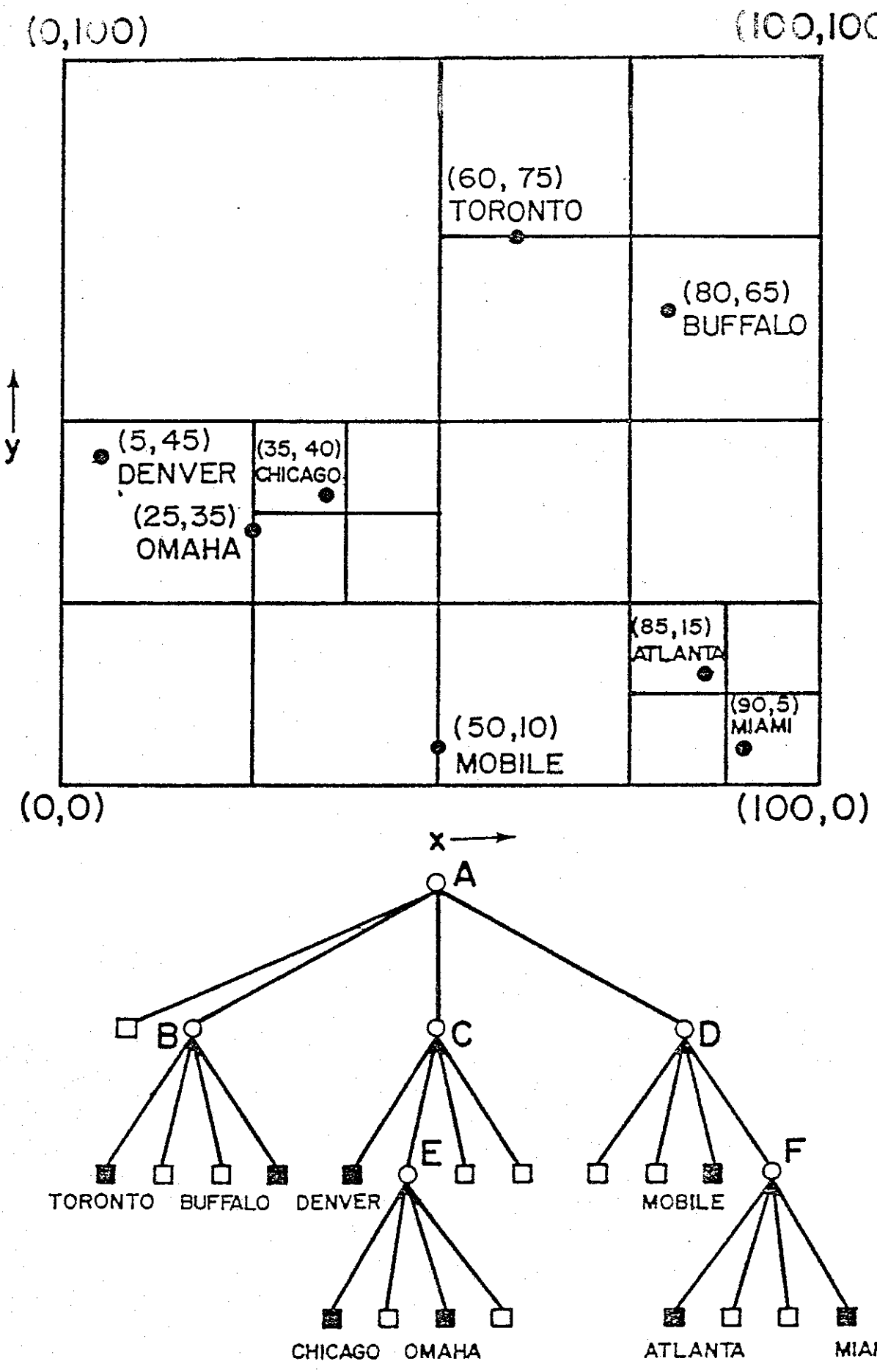


Figure 6.5. A PR quadtree and the records it represents.

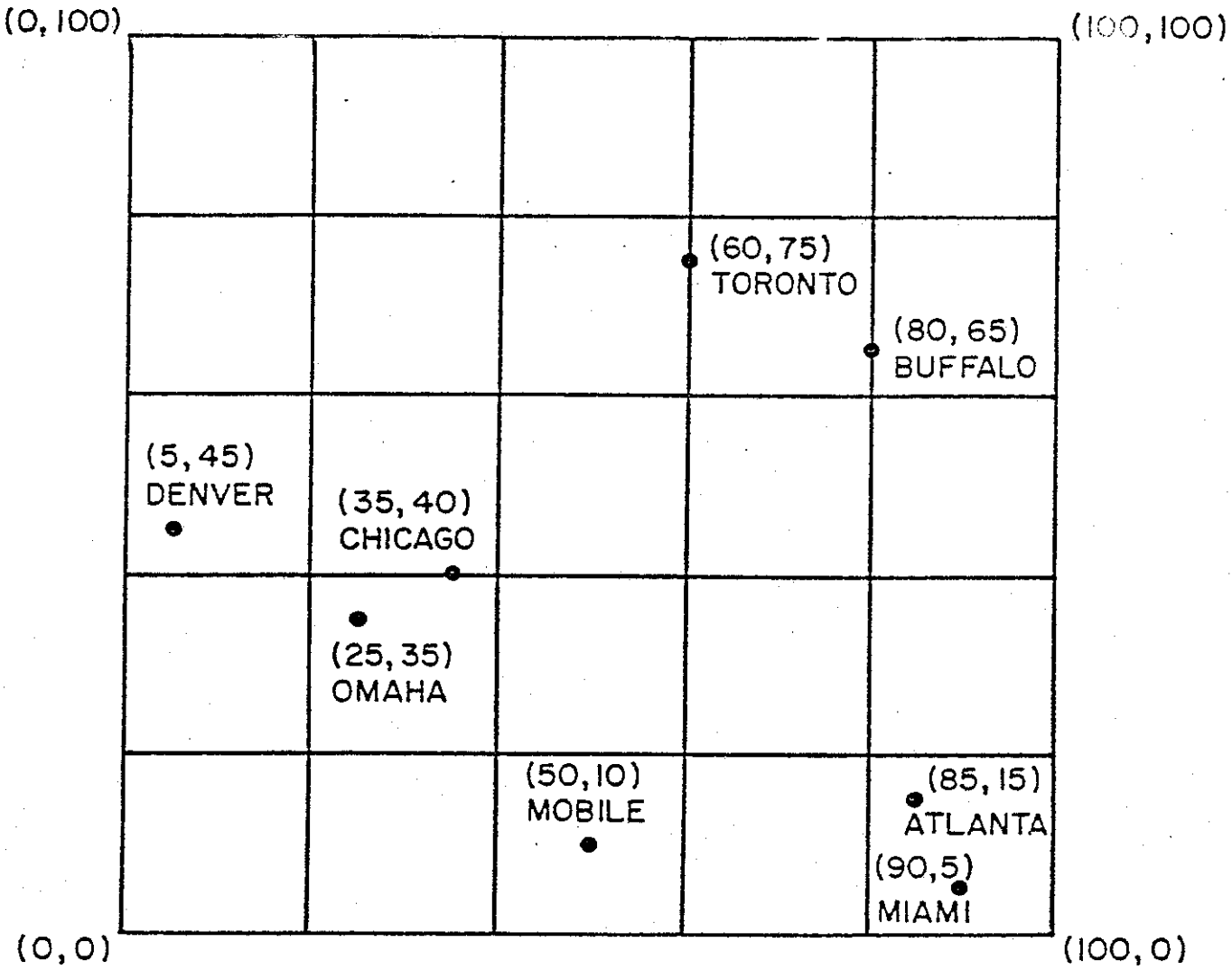


Figure 6.6. Grid representation corresponding to Figure 6.1 with a search radius of 20.

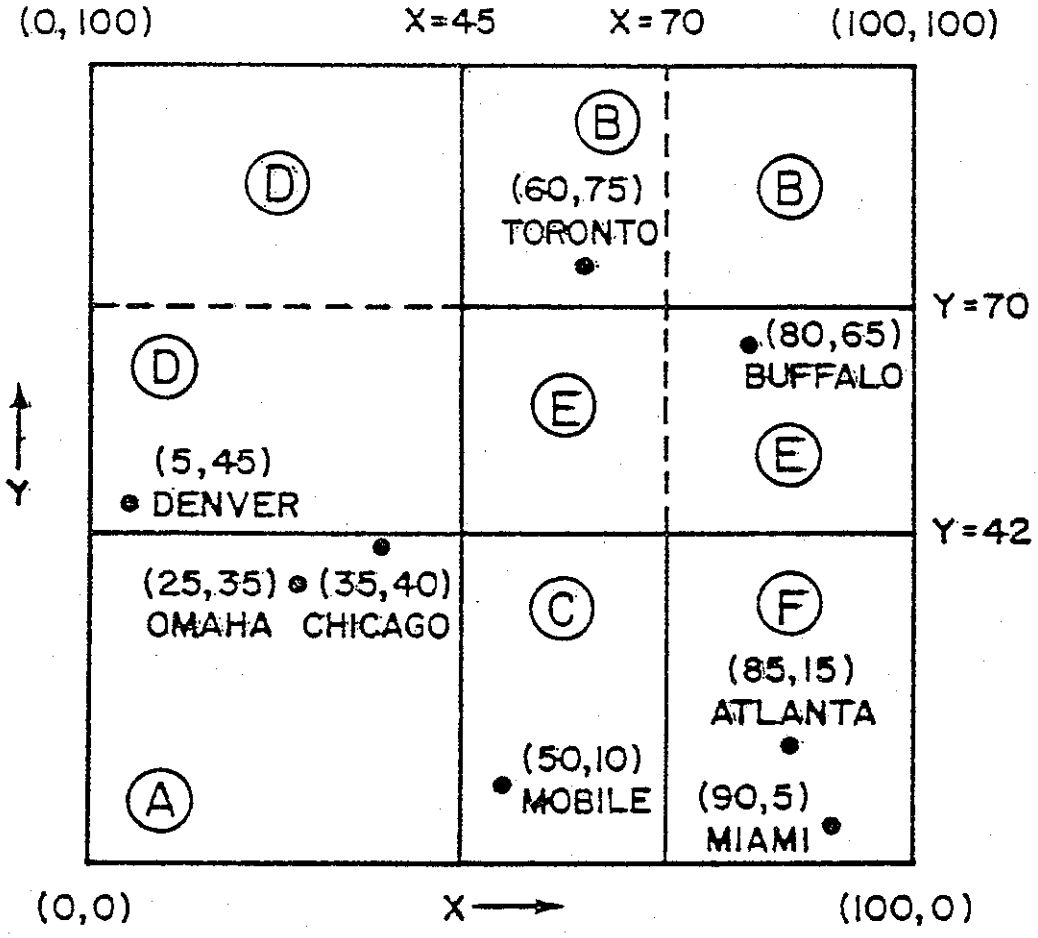


Figure 6.7. Grid directory for the data of Figure 6.1.

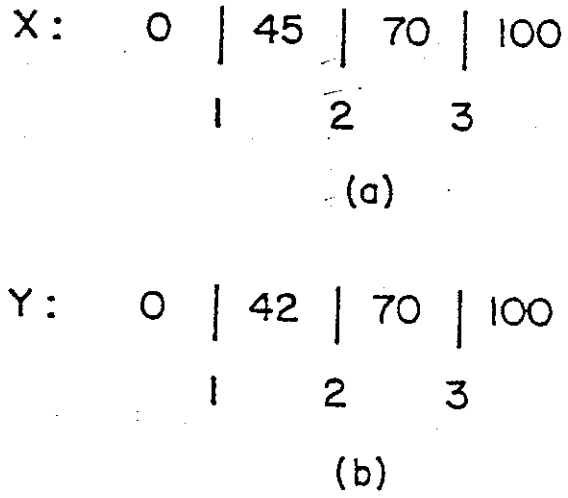


Figure 6.8. Linear scales for (a) x and (b) y corresponding to the grid directory of Figure 6.7.

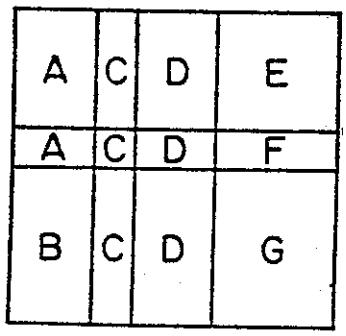


Figure 6.9. Example grid directory illustrating directory merging.

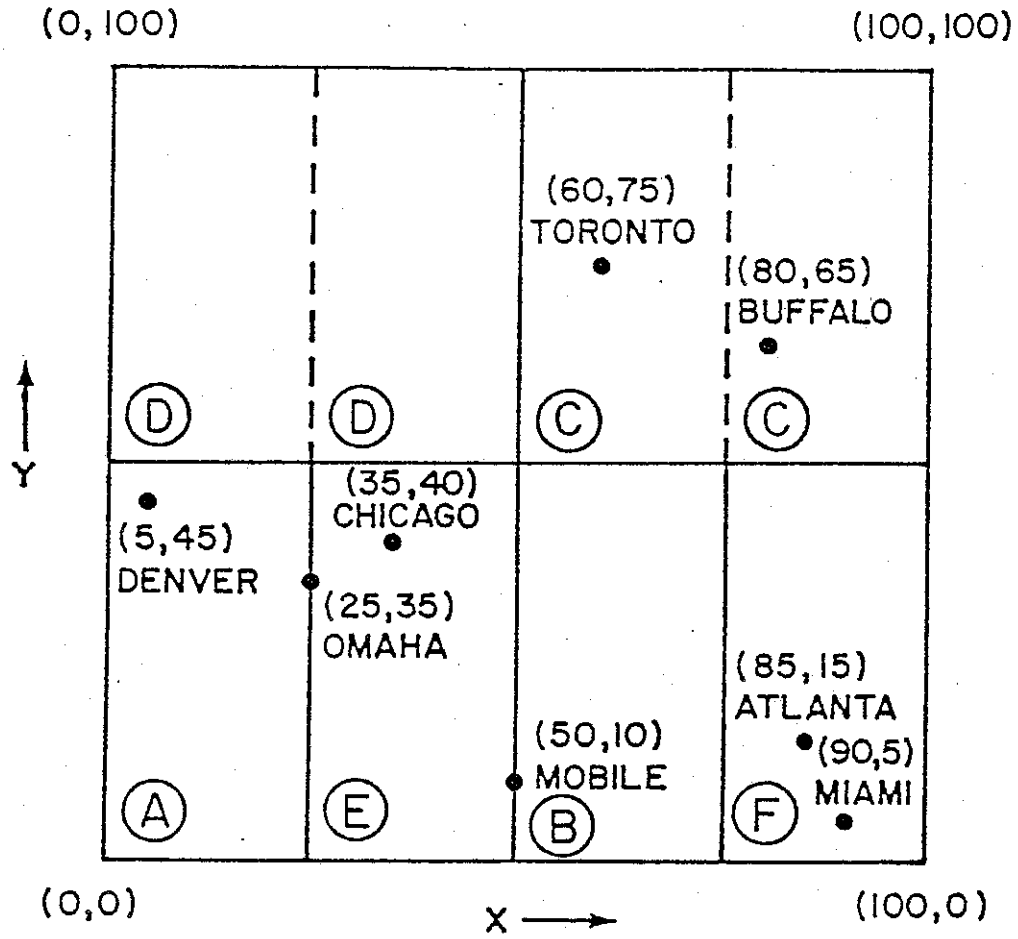


Figure 6.10. EXCELL representation corresponding to Figure 6.1.

7. Curvilinear data

The region quadtree [Klin71] approach to region representation is based on a description of the region's interior. This section focuses on representations that specify borders of regions. This is done in the more general context of data structures for curvilinear data. The simplest representation is the polygon in the form of vectors [Nagy79] which are usually specified in the form of lists of pairs of x and y coordinate values corresponding to their start and end points. One of the most common representations is the chain code [Free74] which is an approximation of a polygon. Other popular representations include raster-oriented methods [Merr73, Peuq79], as well as a combination of vectors and rasters (e.g., vasters [Peuq83]). There has also been a considerable amount of interest recently in hierarchical representations. These are primarily based on rectangular approximations to the data [Ball81, Burt77, Peuc76]. In particular, Burton [Burt77] uses upright rectangles, Ballard [Ball81] uses rectangular strips of arbitrary orientation, and Peucker [Peuc76] uses sets of bands. There also exist methods that are based on a regular decomposition in two dimensions, as reported by Hunter and Steiglitz [Hunt79a], Shneier [Shne81c], and Martin [Mart82]. Note that our primary focus is on the facilitation of set operations and not ease of display, which is a characterization of B-splines and Bezier methods [Cohe80].

In many applications polygons are not unrelated, but together form a partition of the study area (termed a polygonal map). It is possible to use the above representations for each curve that bounds two adjacent regions. However, it is often preferable to represent the complete network of borders with a single hierarchical data structure. Some examples include the line quadtree of Samet and Webber [Same84a], the PM quadtree of Samet and Webber [Same83a], and the edge variant of the EXCELL method of Tamminen [Tamm81]. In order to avoid confusion with the point space formulation of the EXCELL method, we shall use the term *edge-EXCELL*. The remainder of this section elaborates further on the strip tree and also on the representations that are based on a regular decomposition, concluding with a brief comparison of these methods.

7.1. Strip trees

The *strip tree* is a hierarchical representation of a single curve that is obtained by successively approximating segments of it by enclosing rectangles. The data structure consists of a binary tree whose root represents the bounding rectangle of the entire curve. For example, consider Figure 7.1 where the curve between points P and Q, at locations (x_P,y_P) and (x_Q,y_Q) respectively, is modeled by a strip tree. The rectangle associated with the root, A in this example, corresponds to a rectangular strip of maximum width, enclosing the curve, whose sides are parallel to the line joining the endpoints of the curve (i.e., P and Q). Next, this rectangle is decomposed into two parts at one of the points (termed a *splitting point*) on the rectangle that is also part of the curve. There is at least one such splitting point. If there are more, then the decomposition is performed using the point that is at a maximum distance from the line joining the endpoints of the curve. If the curve is both continuous and differentiable at the splitting point, then, of course, the boundaries of the rectangle that pass through these points are tangents to the curve. This splitting process is recursively applied to the two sons until every strip is of a width less than a predetermined value. For Figure 7.1, the

first splitting operation results in the creation of strips B and C. Strip C is further split creating strips D and E at which point the splitting process ceases. Figure 7.2 shows the resulting binary tree. Note that each node in the strip tree is implemented as a record with eight fields. Four fields contain the x and y coordinates of the endpoints, two fields contain pointers to the two sons of the node, and two fields contain information about the width of the strip (i.e., w_L and w_R of Figure 7.1).

Figure 7.1 is a relatively simple example. In order to be able to cope with more complex curves, the notion of a strip tree must be extended. In particular, closed curves (e.g., Figure 7.3) and curves that extend past their endpoints (Figure 7.4) require some special treatment. The general idea is that these curves are enclosed by rectangles which are split into two rectangular strips and from now on the strip tree is used as before. Note that the strip tree concept and the related algorithms are regarded by Ballard as completely expanded down to a primitive level of unit line segments on a discrete grid even when the underlying curves are collinear. In order to be able to handle curves that consist of disconnected segments, strips are classified as either regular or not and a special bit is associated with each strip to indicate its status. Formally, a curve is said to be regular if it is connected and has its endpoints touching the ends of the strip.

Like point and region quadtrees, strip trees are useful in applications that involve search and set operations. For example, suppose we wish to determine whether a road crosses a river. Using a strip tree representation for these features, answering this query means basically performing an intersection of the corresponding strip trees. Three cases are possible as is shown in Figure 7.5. Figures 7.5a and 7.5b correspond to the answers NO and YES respectively while Figures 7.5c requires us to descend further down the strip tree. Other operations that can be performed efficiently by using the strip tree data structure include the computation of the union of two curves, length of a curve, areas of closed curves, intersection of curves with areas, point membership, etc. [Ball81]. In particular, for closed curves that are well-behaved, intersection and point membership are $O(\log n)$ processes where n is the number of points describing the curve. Strip trees are also used by Gaston and Lozano-Perez [Gast84] in robotic tactile recognition and localization.

The strip tree can be characterized as a top-down approach to curve approximation. Burton [Burt77] defines a related structure termed a *BSPR* (Binary Searchable Polygonal Representation) which is a bottom-up approach to curve approximation. Once again, the primitive unit of approximation is a rectangle; however, in the case of the BSPR all rectangles are upright (i.e., they have a single orientation). The curve to be approximated is decomposed into a set of simple sections where each simple section corresponds to a segment of the curve which is monotonic in both the x and y values of the points comprising it. The tree is built by combining pairs of adjacent simple sections to yield compound sections. This process is repeatedly applied until the entire curve is approximated by one compound section. Thus we see that terminal nodes correspond to simple sections and non-terminal nodes correspond to compound sections. For a curve with 2^n simple sections, the corresponding BSPR has n levels.

As an example of a BSPR consider the regular octagon in Figure 7.6a having vertices A-H. It can be decomposed into four simple sections - i.e., ABCD, DEF, FGH, and HA. Figure 7.6b shows a level 1 approximation to the four simple sections consisting of

rectangles AIDN, DJFN, HMFK, and AMHL respectively. Pairing up adjacent simple sections yields compound sections AIJF corresponding to AIDN and DJFN, and AFKL corresponding to HMFK and AMHL (see Figure 7.6c). More pairing yields the rectangle for compound section IJKL (see Figure 7.6d). The resulting BSPR tree is shown in Figure 7.6e. Using the BSPR, Burton shows how to perform point in polygon determination and polygon intersection. These operations are implemented by tree searches and splitting operations.

Both the strip tree and the BSPR share the property of being independent of the grid system in which they are embedded. An advantage of the BSPR representation over the strip tree is the absence of a need for special handling of closed curves. However, the BSPR is not as flexible as the strip tree. In particular, the resolution of the approximation is fixed (i.e., once the width of the rectangle is selected it cannot be varied).

7.2. Methods based on a regular decomposition

Strip tree methods approximate curvilinear data with rectangles. Quadtree methods achieve similar results by use of a collection of disjoint squares having sides whose lengths are powers of two. A number of variants of quadtrees are currently in use, and can be differentiated by the type of data that they are designed to represent. All but the PM quadtree of Samet and Webber [Same83a] and the edge-EXCELL of Tamminen [Tamm81] are pixel-based and yield approximations whose accuracy is constrained, in part, by the resolution of the data they are representing. They can be used to represent both linear and non-linear curves. The latter need not be continuous or differentiable. In contrast, the PM quadtree and the edge-EXCELL yield an exact representation of polygons or collections of polygons.

The *edge quadtree* of Shneier [Shne81] is an attempt to store linear feature information (e.g., curves) for an image (binary and grey-scale) in a manner analogous to that used for storing region information. A region containing a linear feature or part thereof is subdivided into four squares repeatedly until a square is obtained that contains a single curve that can be approximated by a single straight line. Each leaf node contains the following information about the edge passing through it: magnitude (i.e., 1 in the case of a binary image or the intensity in case it is a grey-scale image), direction, intercept, and a directional error term (i.e., the error induced by approximating the curve by a straight line using a measure such as least squares). If an edge terminates within a node, then a special flag is set and the intercept denotes the point at which the edge terminates. Applying this process leads to quadtrees in which long edges are represented by large leaves or a sequence of large leaves. However, small leaves are required in the vicinity of corners or intersecting edges. Of course, many leaves will contain no edge information at all. As an example of the decomposition that is imposed by the edge quadtree, consider Figure 7.7 which is the edge quadtree corresponding to the polygon of Figure 7.8 when represented on a 2^4 by 2^4 grid. Note that the edge quadtree in Figure 7.7 requires fewer blocks than Figure 7.8, which is the representation of the polygon when using the methods of Hunter and Steiglitz [Hunt79a].

Closely related to the edge quadtree is the *least square quadtree* of Martin [Mart82]. In that representation, leaf nodes correspond to regions that contain a single

curve that can be approximated by K (fixed *a priori*) straight lines within a least square tolerance. This enables handling curved lines with greater precision and which have fewer nodes than the edge quadtree. A cruder method is described by Omolayole and Klinger [Omol80] where all parts of the image that contain edge data are repeatedly decomposed until obtaining a 2 by 2 quadrant in which they store template-like representations of the edges. This is quite similar to the MX quadtree, except that the data are edges rather than points. However, it is too low a level of representation in that it does not take advantage of the hierarchical nature of the data structure.

We use a variant of the edge quadtree in our geographic information system. In particular, the edge quadtree is implemented in the form of a variant of the linear quadtree [Garg82]. The edge quadtree is represented as a collection of all the leaf nodes comprising it. As with area and point data each leaf node is encoded by a pair of numbers corresponding to its level and a locational code. The latter is a base 4 number corresponding to a sequence of directional codes that locate the leaf along a path from the root of the quadtree. Each edge quadtree leaf node is implemented as a record containing additional information about the line that passes through the node. In general, a non-WHITE edge quadtree node will contain exactly one line segment which intersects two of the node's edges. However, when two or more lines intersect, an edge quadtree representation requires that decomposition be performed to the pixel level. In such a case the number of lines that intersect at the pixel is recorded as the value of the node.

In our system each edge quadtree leaf node is represented by two 32 bit words. The first word contains the level and the locational code of the leaf node. Images as large as 2^{12} by 2^{12} can be handled. The second word contains information about the leaf node. One bit is used to describe its type (1 for a line segment and 0 for a single pixel or a WHITE node). Two bits indicate, for all non-WHITE nodes, the quadrant in which the node is found relative to its father (termed SONTYPE). One bit is unused. The remaining 28 bits are used to describe the leaf node. For a WHITE node they are 0. If the node corresponds to a single pixel, then it contains the number of lines that pass through the pixel. Otherwise, one line segment passes through the node and its intercepts with the leaf's block are stored here. 14 bits are used for each intercept. Two bits indicate which of the four edges of the block are intersected by the intercept. The remaining 12 bits indicate the distance from a corner of the block to the intercept (the left corner for the north and south edges, and the lower corner for the east and west edges). The SONTYPE field is very important because without it, a general purpose merge routine would merge four brother nodes with identical type information, whereas such a merger should only occur if the identical type information signifies an empty node. For example, consider Figure 7.9 which shows the existence of four brother quadrants with identical line segments passing through them. The same is true for four brother quadrants that contain pixels with the same number of lines passing through each of the four pixels. Of course, this problem can be avoided, in part, for the case of a single line segment by storing absolute addresses for the intercepts (the 12 bits are sufficient). However, now a line's descriptor is not independent of the overall quadtree coordinate system. This is of potential use in a paged environment.

The *line quadtree* of Samet and Webber [Same84a] addresses the issue of hierarchically representing images which are segmented into a number of different regions rather than mere foreground and background as is the case for conventional

quadtrees. In particular, it encodes both the areas of the individual regions and their boundaries in a hierarchical manner. This is in contrast to the region quadtree, which encodes only areas hierarchically and the strip tree which encodes only curves hierarchically. The line quadtree partitions the set of regions (termed a map) via a recursive decomposition technique which successively subdivides the map until obtaining blocks (possibly single pixels) that have no line segments passing through their interiors. With each leaf node, a code is associated that indicates which of its four sides form a boundary (not a partial boundary) of any single region. Thus, instead of distinguishing leaf nodes on the basis of being BLACK or WHITE, boundary adjacency information is used. This boundary information is hierarchical in that it is also associated with non-terminal nodes. In essence, wherever a non-terminal node does not form a T-junction with any of the boundaries of its descendants along a particular side, this side is then marked as being adjacent to a boundary.

As an illustration of a line quadtree, consider the polygonal map of Figure 7.10 whose corresponding line quadtree (i.e., block decomposition) is shown in Figure 7.11. The bold lines indicate the presence of boundaries. Note that the south side of the block corresponding to the root is on a boundary which is also the border of the image. As another example, the western side of the SW son of the root in Figure 7.11 does not indicate the presence of a boundary (i.e., the side is represented by a light line) even though it is adjacent to the border of the image. The problem is that the SW son of the root has its NW and SW sons in different regions, as is signalled by the presence of a T-junction along its western side. Having the boundary information at the non-terminal nodes enables boundary following algorithms to be performed quickly, in addition to facilitating the process of superimposing one map on top of another. Observe also that the line quadtree has the same number of nodes as a conventional quadtree representation of the image. Boundaries of leaf nodes that are partially on the boundary between two regions can have their boundaries reconstructed by examining their neighbors along the shared boundary. For example, the southern side of the NW son of the SW son of the root in Figure 7.11, say A, represents a partial boundary. The exact nature of the boundary is obtained by examining the NE and NW sons of the southern brother of A.

The PM quadtree of Samet and Webber [Same83a] and the edge-EXCELL of Tamminen [Tamm81] are attempts to overcome some of the problems associated with the following three structures: the line quadtree, the edge quadtree, and the quadtree formulation of Hunter and Steiglitz (termed an MX quadtree) for representing polygonal maps (i.e., collections of straight lines). In general, all three of these representations correspond to approximations of a map. The line quadtree is based on the approximation that results from digitizing a polygonal map. For the edge and MX quadtrees, the result is still an approximation because vertices are represented by pixels in the edge quadtree and borders are represented by BLACK pixels in the MX quadtree. Another disadvantage of these three representations is that certain properties of polygonal maps cannot be directly represented by them. For example, it is impossible for five line segments to meet at a vertex. In the case of the edge and MX quadtrees we would have difficulty in detecting the vertex and for the line quadtree the situation cannot be handled because all regions comprising the map must be rectilinear. Note that it is impossible for five rectilinear regions to meet at a point. Other problems include a sensitivity to shift and rotation which may result in a loss of accuracy in the original approximation. Finally, due to their approximate nature, these data structures will most likely require a

considerable amount of storage, since each line is frequently approximated at the pixel level thereby resulting in quadtrees which are fairly deep.

The *PM quadtree* represents a polygonal map by using the PR quadtree discussed in Section 6.2. Each vertex in the map corresponds to a data point in the PR quadtree. We define a *q-edge* to be a segment of an edge of the map that either spans an entire block in the PR quadtree (e.g., segment RS in Figure 7.13) or extends from a boundary of a block to a vertex within the block (i.e., when the block contains a vertex - e.g., segment CV in Figure 7.13).

For every leaf in the PR quadtree we partition all of its q-edges into seven classes. Each of these classes is stored in a balanced binary tree [Aho74]. One class corresponds to the set of q-edges that meet at a vertex within the block's region. This class is ordered in an angular manner. The remaining q-edges that pass through the block's region must enter at one side and exit via another. This yields six classes: NW, NS, NE, EW, SW, and SE, where SW denotes q-edges that intersect both the southern and western boundaries of the block's region. Note that these classes are often empty. The q-edges of these classes are ordered by the position of their intercepts along the perimeter of the block's region. A q-edge that coincides with the border of a leaf's region is placed in either class NS or EW as is appropriate. For example, consider the polygonal map of Figure 7.12 and its corresponding PM quadtree in Figure 7.13. The block containing vertex C has one balanced binary tree for the q-edges intersecting vertex C (three balanced binary tree nodes for q-edges CM, CN, and CV) and one balanced binary tree for the q-edges intersecting the NW boundary (one balanced binary tree node for q-edge ST). In total, the PM quadtree of Figure 7.1v3 contains seven quadtree leaf nodes, nine non-empty balanced binary trees, containing seventeen nodes.

The PM quadtree provides a convenient, reasonably efficient data structure for performing a variety of operations. Point-in-polygon determination is achieved by finding a bordering q-edge with respect to each of the seven classes and then selecting the closest of the seven as the true bordering q-edge. The execution time of this procedure is proportional to the depth of the PM quadtree. It should be noted that the depth of the PM quadtree is inversely proportional to the log of the minimum separation between two vertices plus the log of the number of edges in the polygonal map [Same83c]. Besides point-in-polygon determination, there exist efficient algorithms for insertion of an edge into the map, overlaying two maps, clipping and windowing a map, and range searching (i.e., determining all polygons within a given distance of a point).

The *edge-EXCELL* method is an application of the EXCELL method for point data (described in Section 6.4) to polygonal maps. It is based on a regular decomposition. The principle guiding the decomposition process and the data structure are identical to that used for representing points. The only difference is that now the data consists of straight line segments that intersect the cells (i.e., grid blocks). Once again, a grid directory is used which maps the cells into storage areas of a finite capacity (e.g., buckets) which often reside on disk. As the buckets overflow (i.e., the number of line segments intersecting them exceeds the capacity of the bucket), buckets are split into two equal-sized grid blocks, which may also lead to a doubling in the size of the directory. If the polygonal map contains a vertex at which m lines intersect, and m is greater than the bucket capacity, then no matter how many times the bucket is split it

will be impossible to store all the line segments in one bucket. In such a case, edge-EXCELL makes use of overflow buckets. This is a disadvantage of edge-EXCELL when compared with the PM quadtree.

Using edge-EXCELL, point-in-polygon determination is achieved by a two step process. First, the cell in which the point lies is located. Second, the corresponding polygon is determined by finding the closest polygon border in any given direction by use of a technique known as ray casting [Roth82] (similar to searching for the closest q-edge when using the PM quadtree). Tamminen [Tamm83] has shown that, in practice, this requires on the average little more than one cell access. Edge-EXCELL has also been used to do hidden line elimination [Tamm82].

7.3. Comparison

The chain code is the most compact of the representations. However, it is a very local data structure and is thus rather cumbersome when attempting to perform set operations such as intersection of two curves represented using it. In addition, like the strip tree, and to a lesser extent the BSPR, it is not tied to a particular coordinate system. This is a problem with methods based on a regular decomposition, although it is somewhat reduced for the PM quadtree and edge-EXCELL.

Representations based on a regular decomposition have a number of advantages over the strip tree. First, more than one curve can be represented with one instance of the data structure - a very important feature for maps. Second, they are unique. In contrast, only one curve can be represented by a single strip tree. Also, the strip tree is not unique when the curve is closed, not regular, or contains more than one end point. However, the strip tree is invariant under shifts and rotations. The line quadtree is better than the MX quadtree of [Hunt79a], which represents boundary lines as narrow BLACK regions, for two reasons. First, narrow regions are costly in terms of the number of nodes in the quadtree. Second, it commits the use to a specific thickness of the boundary line which may be unfortunate - e.g., when regenerating the picture on an output device. Also, such arbitrary decisions of representation accuracy are not very appropriate when data is to be stored in a data base.

The edge quadtree representation has a number of problems. First, it uses up too much space for vertices since the decomposition must proceed to the pixel level. Second, the slopes of the various segments through the nodes are inaccurate due to roundoff error induced by the digitization process. The PM quadtree does not have such a problem but is more difficult to integrate into a database containing data of different types due to the large amount of information that must be stored at each terminal node.

At this point it might be appropriate to speculate on some other data structures for curvilinear data. Representations based on regular decomposition are attractive because they enable efficient computation of set operations. In particular, at times it is convenient to perform the operations on different kinds of geometric entities (e.g., intersecting curves with areas, etc.). The strip tree is an elegant data structure from the standpoint of approximation. However, it has the disadvantage that decomposition points are independent of the coordinate system of the image (i.e., they are at arbitrary points dependent on the data rather than at predetermined positions as is the case with

a data structure that is based on a regular decomposition). Thus answering a query such as "Find all wheat growing regions within 20 miles of the Mississippi River" is not easy to do when the river is represented as a strip tree and the wheat growing regions are represented by region quadtrees. The problem is that while quadtree methods merely require pointer chasing operations, strip tree methods may lead to complex geometric computations. What is desired is a regular decomposition strip tree or variant thereof that meets these issues and those raised earlier.

The data structures discussed in this section are rooted in the image processing area and were designed primarily to represent curves and lines. Computational geometry is another area where similar problems arise [Edel84, Tous80]. This is a rapidly changing field having its roots in the work of Shamos and Hoey [Sham75, Sham78] and focusing on problems of asymptotical computational complexity of geometric algorithms. However, a full presentation of this field is beyond the scope of this survey. Nevertheless, in the following we do give a brief sample of the type of results attainable for similar problems. Many of the solutions (e.g., [Lipt77]) are based on the representation of line segments as edges and vertices in a graph.

For example, an alternative to the PM quadtree and edge-EXCELL is the *K-structure* of Kirkpatrick [Kirk83]. It is a hierarchical structure based on triangulation rather than a regular decomposition. The notion of hierarchy in the K-structure is radically different from that of a quadtree, in that instead of replacing a group of triangles by a single triangle at the next higher level, a group of triangles is replaced by a smaller group of triangles. Triangles are grouped for replacement because they share a common vertex. The smaller group results from eliminating the common vertex and then retriangulating. Kirkpatrick [Kirk83] shows that at each level of the hierarchy, at least $\frac{1}{24}$th of the vertices can be eliminated in this manner. The vertices that have been eliminated are guaranteed to have degree of 11 or less, thus bounding the cost of retriangulating. Let v denote the number of vertices in a polygonal map. Then, the size of a K-structure is guaranteed to be $O(v)$ although the worst-case constant of proportionality is 24 times the amount of information stored at a node. It also leads to an $O(\log v)$ query time for point-in-polygon determination. The construction process has a worst-case execution time of $O(v)$ for a triangular subdivision and $O(v \cdot \log v)$ for a general one. The latter is dominated by the cost of triangulating the original polygonal map [Hert83]. Since a triangulation constitutes a convex map, i.e., a planar subdivision formed of convex regions, the work of Nievergelt and Preparata [Niev82] is relevant. They show that the cost of performing a map overlay operation is $O(v \cdot \log v + s)$ where s is the number of intersections of all line segments in the two maps. Finally, it is worth noting that the hierarchical nature of the K-structure may lead to an efficient range-searching algorithm.

In comparing the K-structure with the PM quadtree (and to some extent edge-EXCELL) the qualitative comparison is analogous to that of a point quadtree with a PR quadtree. Both structures have their place; the one to use depends on the nature of the data and the importance of guaranteed worst-case performance. The K-structure organizes the data to be stored while the PM quadtree organizes the embedding space from which the data is drawn. The K-structure has better worst case execution time bounds for similar operations compared to those considered for the PM quadtree. However, considerations such as ease of implementation and integration with representations of other

data types must also be taken into account in making an evaluation. In the case of dynamic files, at present, it would seem to be more convenient to use the PM quadtree since a general updating algorithm for the K-structure has not been reported.

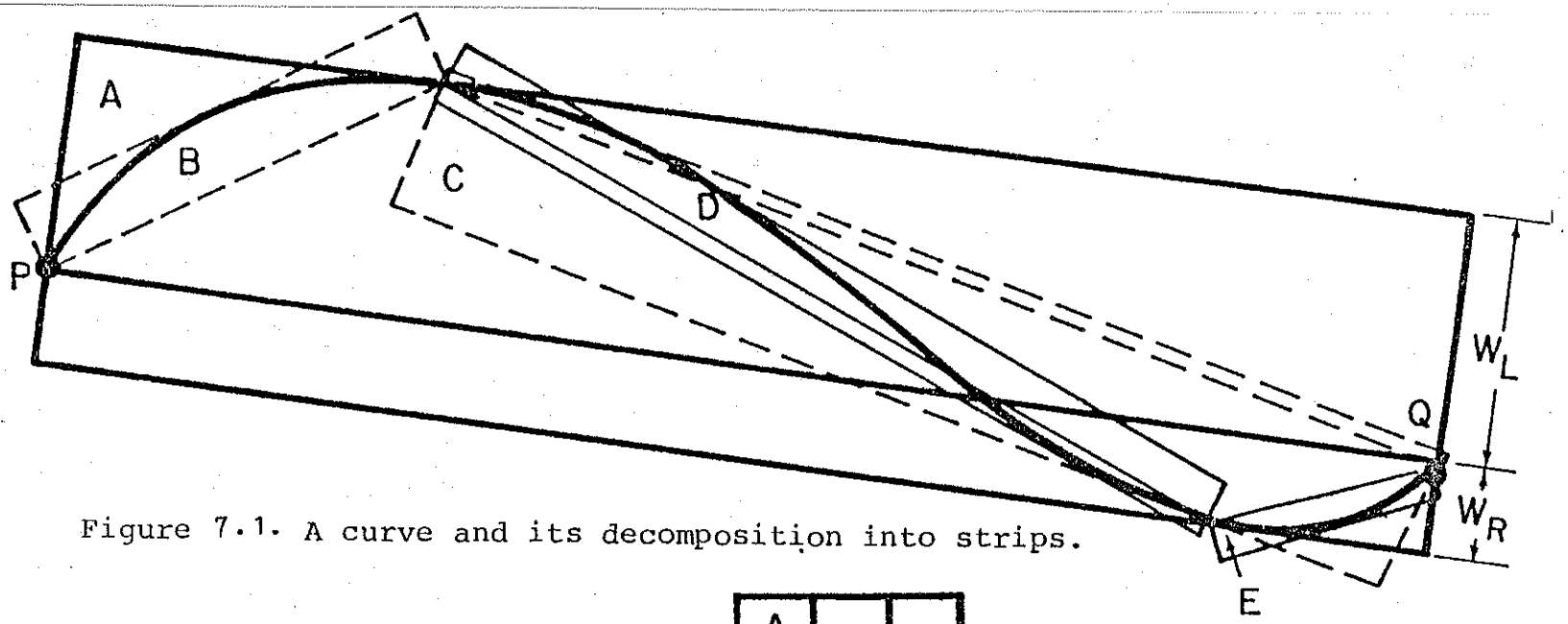


Figure 7.1. A curve and its decomposition into strips.

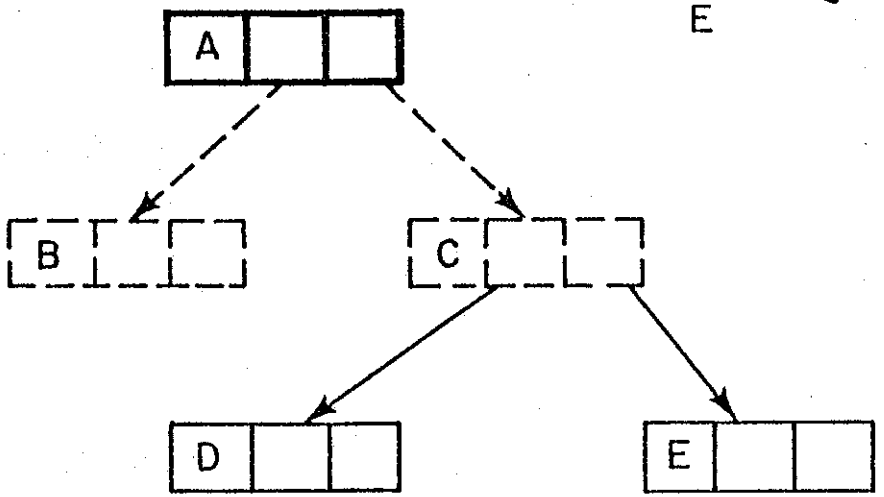


Figure 7.2. Strip tree corresponding to Figure 7.1.

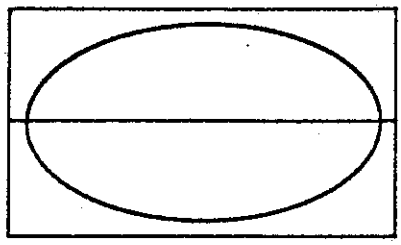


Figure 7.3. Modeling a closed curve by a strip tree.

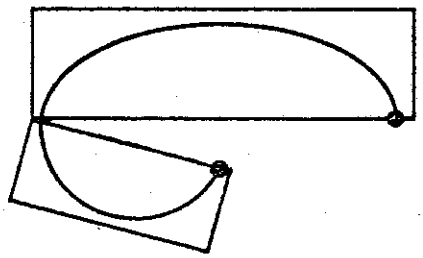


Figure 7.4. Modeling a curve that extends past its endpoints by a strip tree.

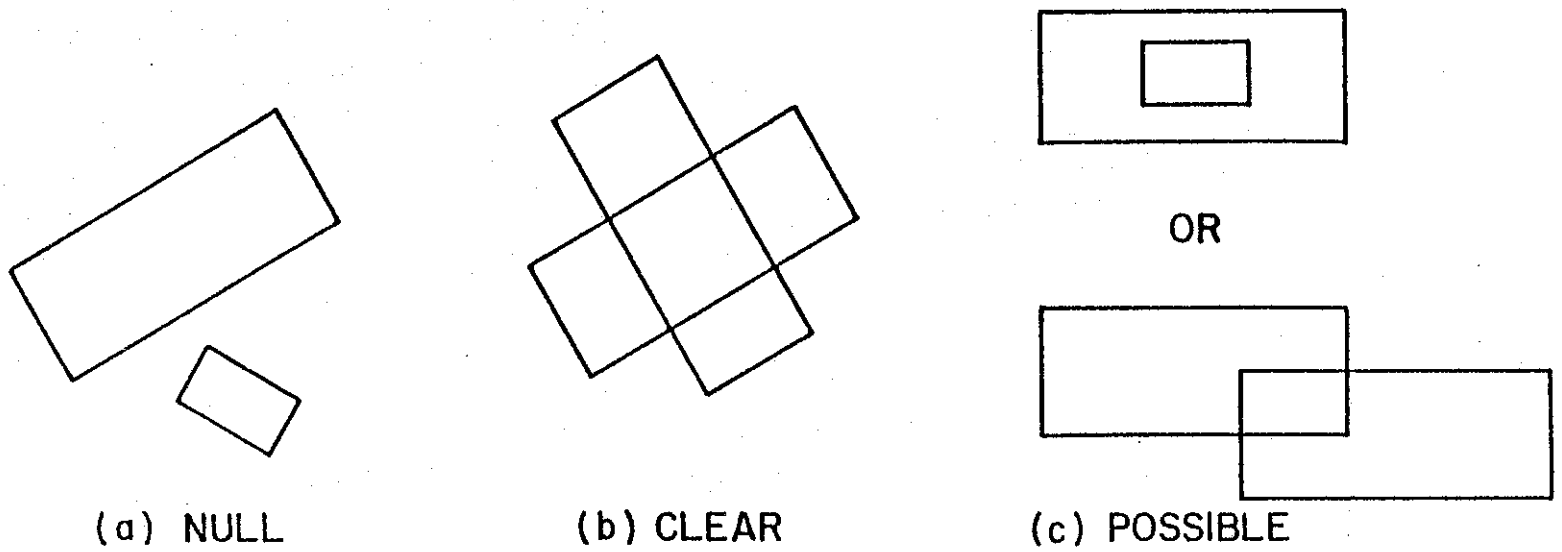


Figure 7.5. Three possible results of intersecting two strip trees.

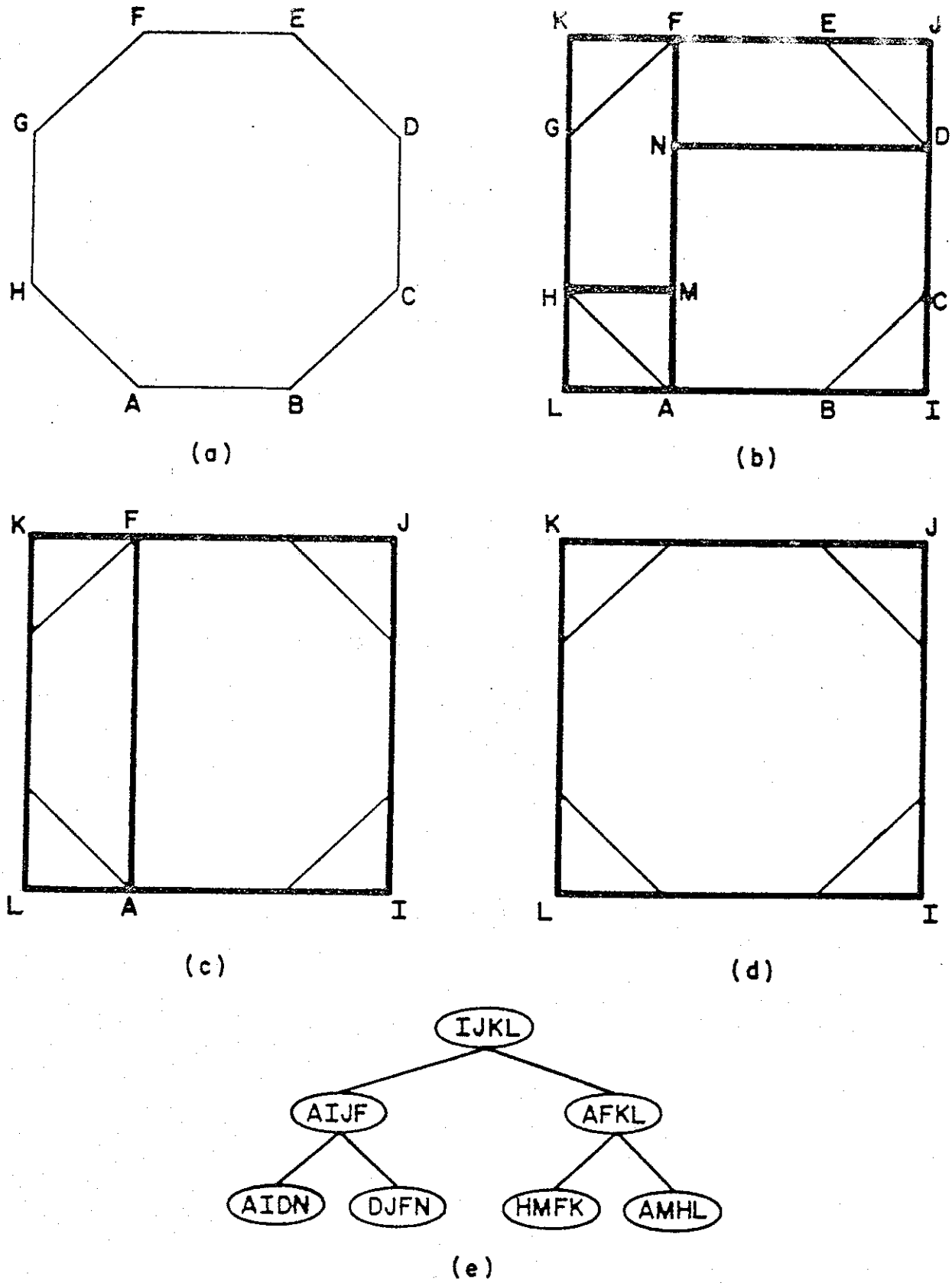


Figure 7.6. (a) A regular octagon and (b)-(d) the three successive approximations resulting from the use of BSPR. (e) The resulting BSPR.

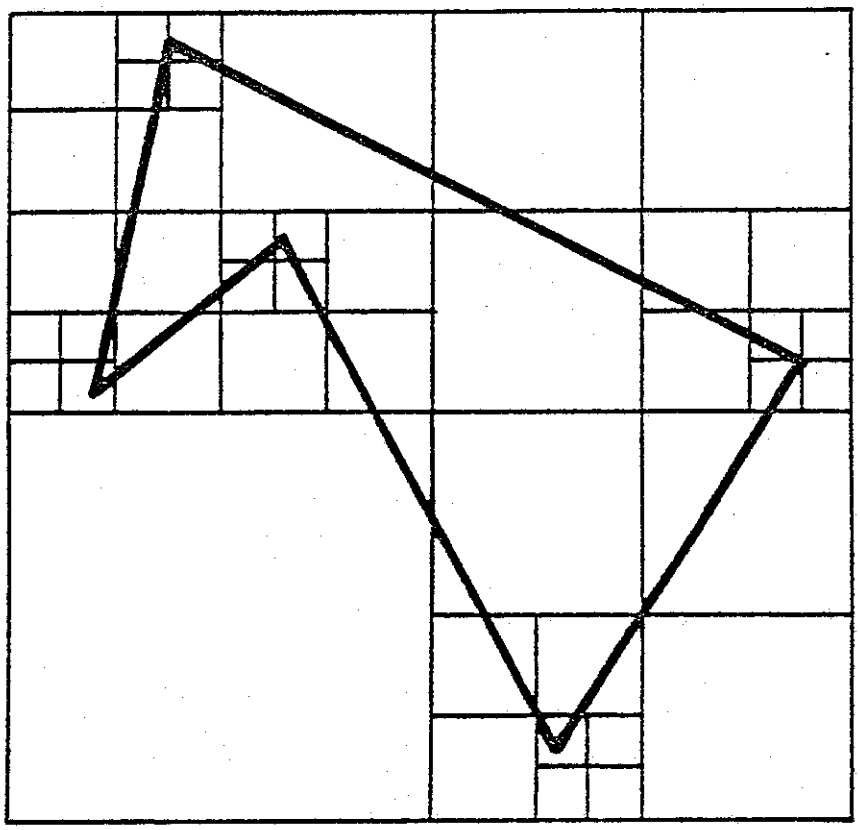


Figure 7.7. The edge quadtree corresponding to the polygon of Figure 7.8.

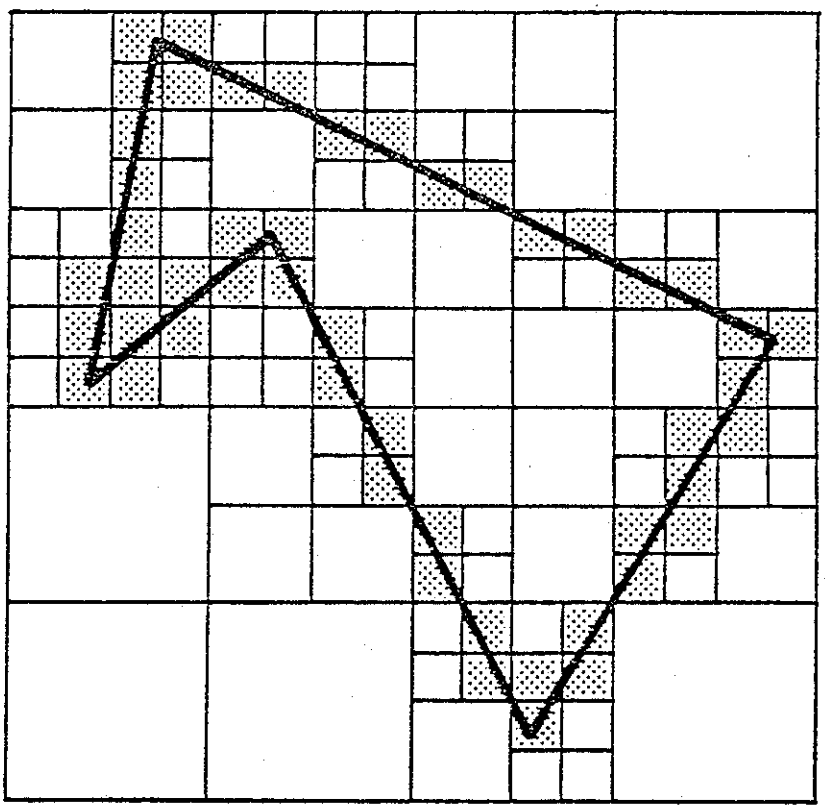


Figure 7.8. Hunter and Steiglitz's quadtree representation of a polygon.

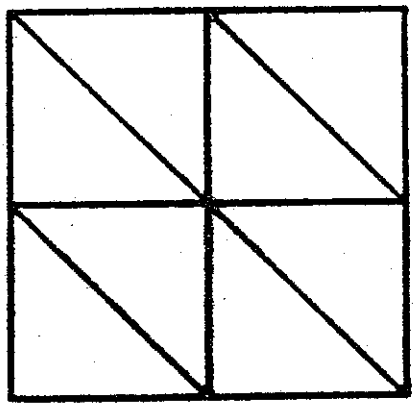


Figure 7.9. An example of four identical siblings.

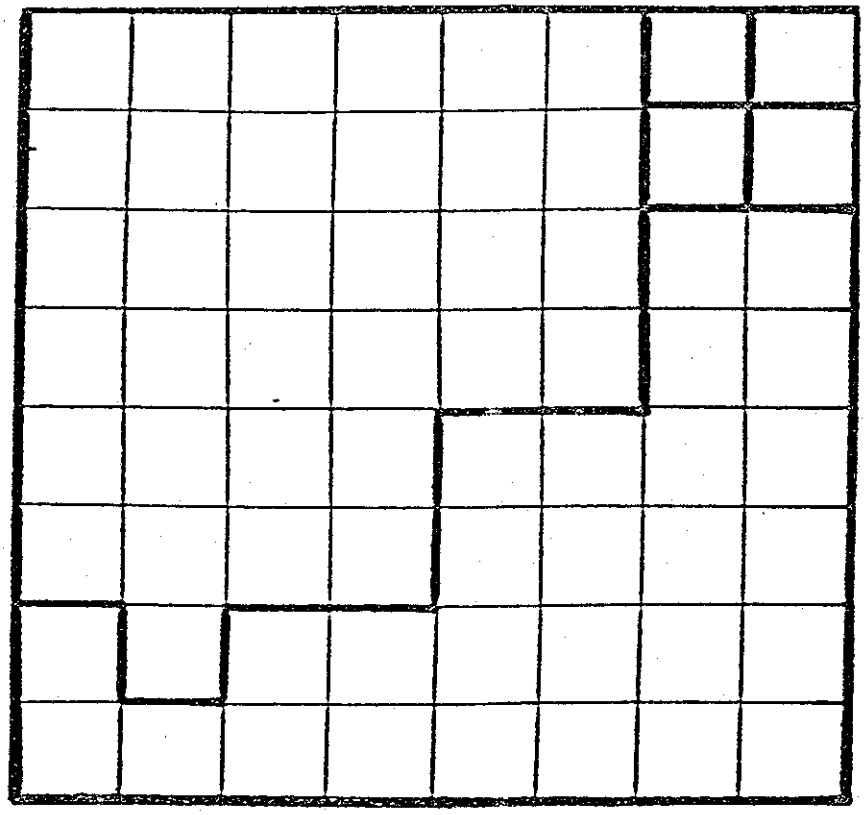


Figure 7.10. Example polygonal map to illustrate line quadtrees.

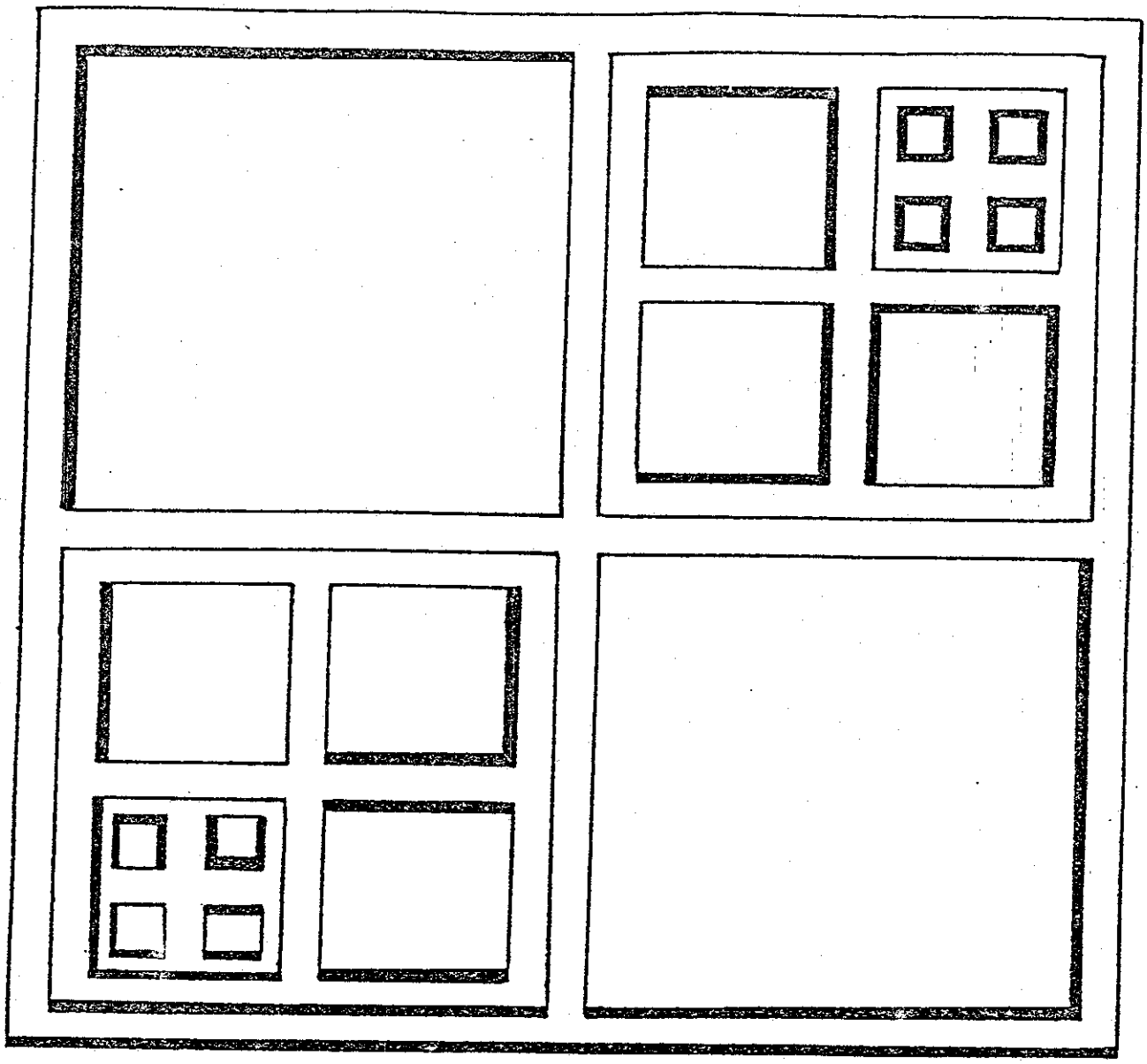


Figure 7.11. Line quadtree corresponding to Figure 7.10.

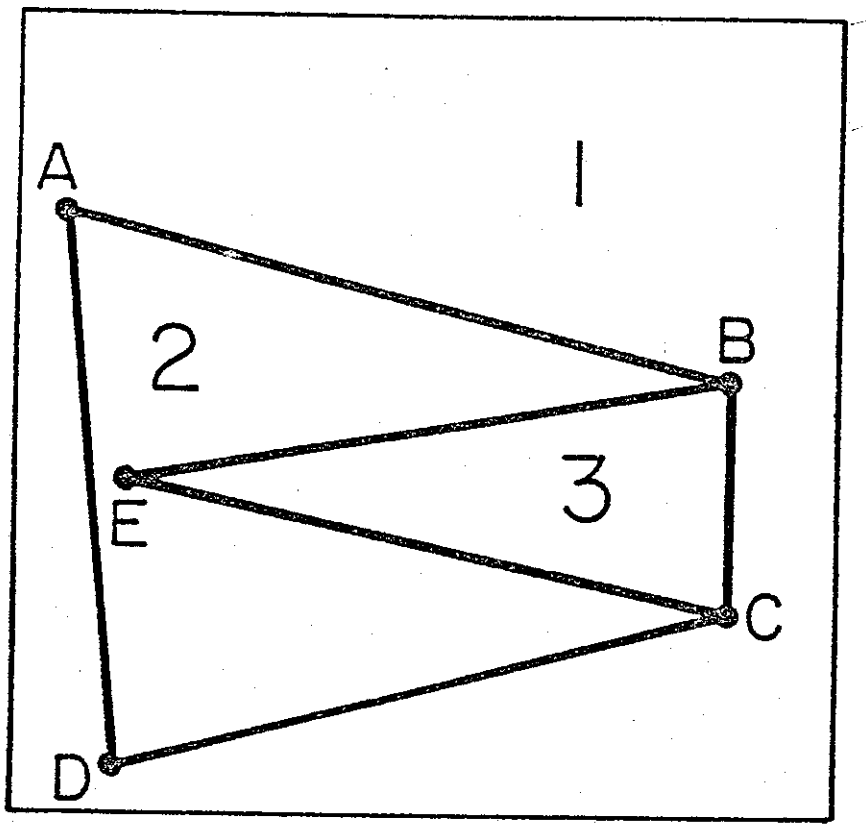


Figure 7.12. Example polygonal map to illustrate PM quadtrees.

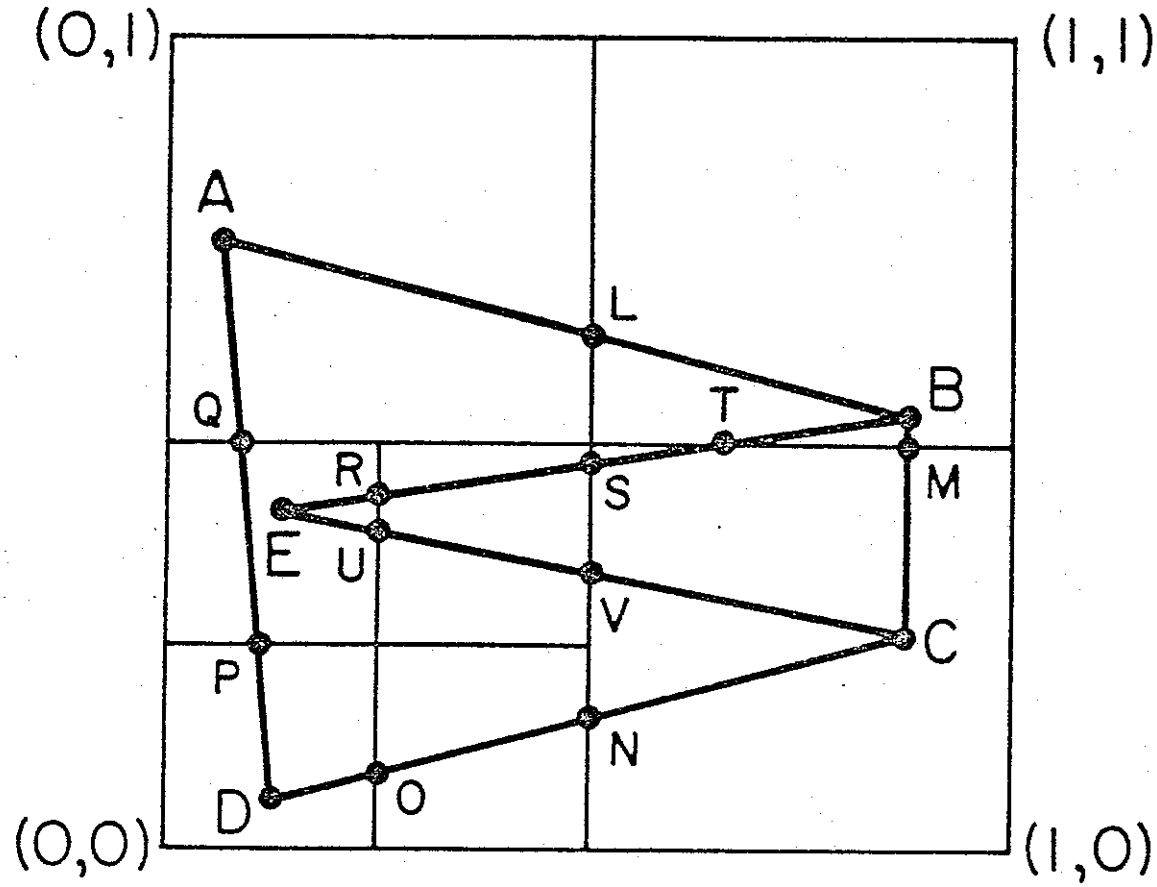


Figure 7.13. PM quadtree corresponding to Figure 7.12.

8. Conclusions and Future Plans

During Phase III of our investigation into the application of hierarchical data structures to geographic information systems, the database system was enhanced in a number of ways. A method was developed for storing information describing individual maps through use of the DMA Standard Lineal Format header file. A method was also developed for storing information about polygons and classes of polygons through use of the attribute attachment functions. Our implementation also allows for the sharing of attribute class definitions among sets of maps. Finally, new functions and improved algorithms have extended the usefulness of the present database system.

Our main topic of research during the next phase of this project will be an examination of other hierarchical data structures for use within our system. Particular emphasis will be placed on structures for representing line data. The structures to be investigated will be largely drawn from those reported on in Sections 6 and 7 of this report. Other probable topics include new work on a database query language, and extension of the memory management system to allow greater flexibility in the node size of the structures used to store maps.

9. References

1. [Aho74] - A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

2. [Ande83] - D.P. Anderson, Techniques for reducing pen plotting time, *ACM Transactions on Graphics 2*, 3(July 1983), 197-212.

3. [Ball81] - D.H. Ballard, Strip trees: A hierarchical representation for curves, *Communications of the ACM 24*, 5(May 1981), 310-321 (see also corrigendum, *Communications of the ACM 25*, 3(March 1982), 213).

4. [Bent75a] - J.L. Bentley and D.F. Stanat, Analysis of range searches in quad trees, *Information Processing Letters 3*, 6(July 1975), 170-173.

5. [Bent75b] - J.L. Bentley, A survey of techniques for fixed radius near neighbor searching, SLAC Report No. 186, Stanford University, Stanford, CA, August 1975.

6. [Bent75c] - J.L. Bentley, Multidimensional binary search trees used for associative searching, *Communications of the ACM 18*, 9(September 1975), 509-517.

7. [Brod66] - P. Brodatz, *Textures*, Dover, New York, 1966.

8. [Burk83] - W.A. Burkhardt, Interpolation-based index maintenance, *BIT 23*, 3(1983), 274-294.

9. [Burt77] - W. Burton, Representation of many-sided polygons and polygonal lines for rapid processing, *Communications of the ACM 20*, 3(March 1977), 166-171.

10. [Cohe80] - E. Cohen, T. Lyche, and R. Riesenfeld, Discrete B-splines and subdivision techniques in computer-aided geometric design and computer graphics, *Computer Graphics and Image Processing 14*, 3(October 1980), 87-111.

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

12. [DeCo76] - F. DeCoulon and U. Johnsen, Adaptive block schemes for source coding of black-and-white facsimile, *Electronics Letters 12*, 3(1976), 61-62 (see also erratum, *Electronics Letters 12*, 6(1976), 152).

13. [DMA83] - Defense Mapping Agency Standard Linear Format (SLF) for digital cartographic feature data, unpublished document, 16 May 1983.

14. [Dyer80] - C.R. Dyer, A. Rosenfeld, and H. Samet, Region representation: boundary codes from quadtrees, *Communications of the ACM 23*, 3(March 1980), 171-179.

15. [Dyer82] - C.R. Dyer, The space efficiency of quadtrees, *Computer Graphics and Image Processing 19*, 4(August 1982), 335-348.

16. [Edel84] - H. Edelsbrunner, L.J. Guibas, and J. Stolfi, Optimal point location in a monotone subdivision, to appear.

17. [Fagi79] - R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, Extendible hashing - a fast access method for dynamic files, *ACM Transactions on Database Systems 4*, 3(September 1979), 315-344.

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

19. [Free74] - H. Freeman, Computer processing of line-drawing images, *ACM Computing Surveys 6*, 1(March 1974), 57-97.

20. [Frie77] - J.H. Friedman, J.L. Bentley, and R.A. Finkel, An algorithm for finding best matches in logarithmic expected time, *ACM Transactions on Mathematical Software 3*, 3(September 1977), 209-226.

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

22. [Gast84] - P.C. Gaston and T. Lozano-Perez, Tactile recognition and localization using object models: the case of polyhedra on a plane, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 3(May 1984), 257-266.

23. [Hert83] S. Hertel and K. Mehlhorn, Fast triangulation of simple polygons, *Proceedings of the 1983 International FCT Conference*, Borgholm, Sweden, August 1983, 207-218 (Lecture Notes in Computer Science 158, Springer Verlag, New York, 1983).

24. [Hunt79] - G.M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

25. [Kirk83] - D. Kirkpatrick, Optimal search in planar subdivisions, *SIAM Journal on Computing 12*, 1(February 1983), 28-35.

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

27. [Knot71] - G.D. Knott, Expandable open addressing hash table storage and retrieval, *Proceedings of SIGFIDET Workshop on Data Description, Access, and Control*, San Diego, CA, November 1971, 187-206.

28. [Knut73] - D.E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

29. [Knut75] - D.E. Knuth, *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1975.

30. [Lauz84] - J.P. Lauzon, D.M. Mark, L. Kikuchi, and J.A. Guevara, Two-dimensional

run-encoding for quadtree representation, Department of Geography, State University of New York at Buffalo, Buffalo, NY, 1984.

31. [Lee77] - D.T. Lee and C.K. Wong, Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees, *Acta Informatica 9*, 1(1977), 23-29.

32. [Lete83] - P. Letelier, Transmission d'images a bas debit pour un systeme de communication telephonique adapte aux sourds, These de 3-eme cycle, Universite de Paris-Sud, Paris, September 1983.

33. [Li82] - M. Li, W.I. Grosky, and R. Jain, Normalized quadtrees with respect to translations, *Computer Graphics and Image Processing 20*, 1(September 1982), 72-81.

34. [Linn73] - J. Linn, General methods for parallel searching, Technical Report 81, Digital Systems Laboratory, Stanford University, Stanford, CA, May 1973.

35. [Litw80] - W. Litwin, Linear hashing: a new tool for file and table addressing, *Proceedings of the Sixth International Conference on Very Large Data Bases*, Montreal, Canada, October 1980, 212-223.

36. [Mart82] - J.J. Martin, Organization of geographical data with quad trees and least square approximation, *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, Las Vegas, NV, 1982, 458-463.

37. [Mats84] - T. Matsuyama, L.V. Hao, and M. Nagao, A file organization for geographic information systems based on spatial proximity, *Computer Vision, Graphics, and Image Processing 26*, 3(June 1984), 303-318.

38. [Merr78] - T.H. Merrett, Multidimensional paging for efficient database querying, *Proceedings of the International Conference on Management of Data*, Milan, Italy, June 1978, 277-289.

39. [Merr82] - T.H. Merrett and E.J. Otoo, Dynamic multipaging: a storage structure for large shared data banks, in *Improving Database Usability and Responsiveness*, P. Scheuermann, Ed., Academic Press, New York, 1982, 237-254.

40. [Merr73] - R.D. Merrill, Representations of contours and regions for efficient computer search, *Communications of the ACM 16*, 2(February 1973), 69-82.

41. [Mort66] - G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM Canada, 1966.

42. [Nagy79] - G. Nagy and S. Wagle, Geographic data processing, *ACM Computing Surveys 11*, 2(June 1979), 139-181.

43. [Niev82] - J. Nievergelt and F.P. Preparata, Plane-sweep algorithms for intersecting geometric figures, *Communications of the ACM 25*, 10(October 1982), 739-746.

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

45. [Omol80] - J.O. Omolayole and A. Klinger, A hierarchical data structure scheme for storing pictures, in *Pictorial Information Systems*, S.K. Chang and K.S. Fu, Eds., Springer Verlag, Berlin, 1980.

46. [ORou81] - J. O'Rourke, Dynamically quantized spaces for focusing the Hough Transform, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Vancouver, Canada, August 1981, 737-739.

47. [ORou84] - J. O'Rourke and K.R. Sloan, Jr., Dynamic quantization: two adaptive data structures for multidimensional squares, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 3(May 1984), 266-280.

48. [Oren82] - J.A. Orenstein, Multidimensional tries used for associative searching, *Information Processing Letters 14*, 4(June 1982), 150-157.

49. [Oren83] - J.A. Orenstein, A dynamic hash file for random and sequential accessing, *Proceedings of the Sixth International Conference on Very Large Data Bases*, Florence, Italy, October 1983, 132-141.

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

51. [Ouks83] - M. Ouksel and P. Scheuermann, Storage mappings for multidimensional linear dynamic hashing, *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Atlanta, GA, March 1983, 90-105.

52. [Peuc76] - T. Peucker, A theory of the cartographic line, *International Yearbook of Cartography*, 1976.

53. [Peuq79] - D.J. Peuquet, Raster processing: an alternative approach to automated cartographic data handling, *American Cartographer* (April 1979), 129-239.

54. [Peuq83] - D.J. Peuquet, A hybrid data structure for the storage and manipulation of very large spatial data sets, *Computer Vision, Graphics, and Image Processing 24*, 1(October 1983), 14-27.

55. [Robi81] - J.T. Robinson, The k-d-B-tree: a search structure for large multidimensional dynamic indexes, *Proceedings of the SIGMOD Conference*, Ann Arbor, MI, April 1981, 10-18.

56. [Rose82] - A. Rosenfeld, H. Samet, C. Shaffer, and R.E. Webber, Application of hierarchical data structures to geographical information systems, Computer Science TR-1197, University of Maryland, College Park, MD, June 1982.

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

58. [Roth82] - S.D. Roth, Ray casting for modeling solids, *Computer Graphics and Image Processing 18*, 2(February 1982), 109-144.

59. [Same80a] - H. Samet, Region representation: quadtrees from boundary codes, *Communications of the ACM 23*, 3(March 1980), 163-170.

60. [Same80b] - H. Samet, Deletion in two-dimensional quad trees, *Communications of the ACM 23*, 12(December 1980), 703-710.

61. [Same81] - H. Samet, Connected component labeling using quadtrees, *Journal of the ACM 28*, 3(July 1981), 487-501.

62. [Same82] - H. Samet, Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing 18*, 1(January 1982), 37-57.

63. [Same83a] - H. Samet and R. E. Webber, Using quadtrees to represent polygonal maps, *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, 127-132 (also University of Maryland Computer Science TR-1372).

64. [Same83b] - H. Samet and M. Tamminen, Computing geometric properties of images represented by linear quadtrees, University of Maryland Computer Science TR-1359, December 1983.

65. [Same83c] - H. Samet and E. V. Krishnamurthy, A quadtree-based matrix manipulation system, in progress, 1983.

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

67. [Same84b] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984) (also University of Maryland Computer Science TR-1329).

68. [Same84c] - H. Samet and C.A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, University of Maryland Computer Science TR-1432, August 1984.

69. [Sham75] - M.I. Shamos and D. Hoey, Closest-point problems, *Proceedings of the Sixteenth Annual IEEE Symposium on the Foundations of Computer Science*, Berkeley, CA, October 1975, 151-162.

70. [Sham78] - M.I. Shamos, Computational geometry, Ph.D. dissertation, Department of Computer Science, Yale University, New Haven, CT, 1978.

71. [Shne81] - M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Computer Graphics and Image Processing 17*, 3(November 1981), 211-224.

72. [Sloa81] - K.R. Sloan, Jr., Dynamically quantized pyramids, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Vancouver, Canada, August 1981, 734-736.

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

74. [Tamm82] - M. Tamminen, Hidden lines using the EXCELL method, *Computer Graphics Forum 11*, 3, 1982, 96-105.

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

76. [Tani75] - S. Tanimoto and T. Pavlidis, A hierarchical data structure for picture processing, *Computer Graphics and Image Processing 4*, 2(June 1975), 104-119.

77. [Tous80] - G.T. Toussaint, Pattern recognition and geometrical complexity, *Proceedings of the Fifth International Conference on Pattern Recognition*, Miami Beach, FL, December 1980, 1324-1346.

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

79. [vanL81] - J. van Leeuwen and D. Wood, The measure problem for rectangular ranges in d-space, *Journal of Algorithms 2*, 3(September 1981), 282-300.

80. [Will82] - D.E. Willard, Polygon retrieval, *SIAM Journal on Computing 11*, 1(February 1982), 149-165.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ETL-0376	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: Contract Report 9-27-83 - 9-26-84	
4. TITLE *(and Subtitle)*: APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS (PHASE III)	5. TYPE OF REPORT & PERIOD COVERED: Final Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Hanan Samet; Azriel Rosenfeld	8. CONTRACT OR GRANT NUMBER(s): DAAK70-81-C-0059
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Vision Laboratory, University of Maryland, College Park, MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Engineer Topographic Laboratories, Fort Belvoir, VA 22060-5546	12. REPORT DATE: November 1984
	13. NUMBER OF PAGES: 114
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

The previous reports completed under this contract are ETL-0301 (AD-A124 196) and ETL-0337 (AD-A134 999).

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

geographical information system
data strucure
quadtrees
region-base quadtrees
point quadtrees
attribute attachment

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This document is report of Phase III of an investigation of the application of hierarchical data structure to geographical information systems. It deals primarily with enhancements and improvements to the information system package, an evaluation of design decisions and the collection of empirical results to indicate the utility of the software and justify the design decisions. Tasks reported on include: Attribute attachment, DMA SLF compatibility, memory management improvements and data base enhancements.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*