A File Organization Scheme for Polygon Data

Chung Hee Hwang & Wayne A. Davis

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract[1]

This paper presents a file organization scheme for representing polygon data by a quadtree. The proposed scheme is an adaptive cell method that is based on *extendible hashing* and *interpolation-based index maintenance*. It aims, on the average, to locate the record associated with a given key with one disk access (or at most two), maintaining a high storage utilization ratio. It also aims to process range search and set operations efficiently. The dynamic file organization capabilities of the scheme and the algorithm for range search are described.

1. Introduction

Given a sequence of k points $P_i=(x_i,y_i)$, for $1 \le i \le k$, in a plane, a polygon with vertices P_i is the sequence of line segments, called edges, P_1P_2, P_2P_3,..., P_kP_1. If these k edges do not have any intersection points, the polygon is simple. A simple polygon divides the plane into two distinct regions. The interior of a simple polygon is called a *polygonal region*. A *complex polygonal region* is a polygonal region which is allowed to have one or more holes in it. Hereafter, a complex polygonal region is called a *polygon*. A *polygon network* for a study area is a set of disjoint polygons overlapping the study area such that the set of polygons yields a total partition of the study area. Each polygon in a polygon network has a unique name.

Although polygon networks have been traditionally represented in vector format, recently quadtree encoding of a polygon network has received increased attention. The quadtree [7] is a dynamic data structure developed to reduce the storage requirement of raster representation by aggregating homogeneous cells. Nevertheless, as the original quadtree concept was based on the assumption that quadtrees were resident in main memory, quadtree structures may not be directly applicable to data resident in external memory. For example, the need to follow pointers may lead to a larger number of page faults than are acceptable in an interactive environment.

In an effort to overcome the frequent page fault problem, there have been studies to represent a quadtree as a linear quadtree [5] and use a B-tree file structure in organizing the data [1,8]. While the B-tree organization of a linear quadtree is a significant improvement over the original quadtree organization in the expected number of disk accesses for single record retrieval, the absence of a localization property is a primary disadvantage. A query usually requires the whole file to be retrieved even though the query can be answered with information from a small part of the file. For example, range search is very awkward and set operations are not efficient because there is no implied connection between data buckets in physical storage and regions in the search space. Furthermore, the B-tree organization of quadtree encoded data still needs several disk accesses to retrieve each record because it is essentially a tree that is accessible with $O(\log n)$ I/O operations, where n is the number of records in the file.

From this perspective, a file organization scheme is developed for polygon networks encoded as a linear quadtree with an aim to locate the record associated with a given key with an average of one disk access (or at most two), maintaining a high storage utilization ratio. Furthermore, the following types of spatial queries are to be supported efficiently: the point-in-polygon query, range search and set operations such as polygon union or intersection and polygon overlay. The scheme is an adaptive cell method and it is based on extendible hashing [4] and interpolation hashing [2], a k-dimensional generalization of linear hashing [6].

2. Definitions and Notation

Let the study area, $\mathbf{U} = [0, 2^n)^2$, be an image of $2^n \times 2^n$ unit square pixels that intersects a polygon network, and let each of the pixels have a polygon name (hereafter called *color*) associated with it. Furthermore, let the polygon network on $\mathbf{U}$ be represented by a region quadtree. To yield an arbitrary but consistent total ordering among the blocks of a quadtree, the following hash function is introduced:

Definition 1. Let $(x,y) \in \mathbf{U} = [0,2^n)^2$ be the x and y coordinates of the lower-left corner of a block of a quadtree defined on $\mathbf{U}$ and have the following binary representation:

$$x = \sum a_i 2^i, \text{ and } y = \sum b_i 2^i, \text{ for } 0 \le i \le n-1,$$

where $a_i, b_i \in \{0,1\}$. Then, an *order preserving hash function*, s, that maps (x,y) onto the key space of $[0,4^n)$ is defined by:

$$s(x,y) = \sum(a_i 2^{2i+1} + b_i 2^{2i}), \text{ for } 0 \le i \le n-1.$$

Notice that the key produced for each of the blocks in this manner is essentially the same as the locational code of the block in linear quadtree encoding [5]. Now, a file that represents a polygon network by a quadtree is defined:

Definition 2. A file $\mathbf{F}$ representing a polygon network for the study area $\mathbf{U}$ by a quadtree is the set,

$$\mathbf{F} = \{(K(L),S(L),C(L)): L \in R\},$$

where R is the set of all leaf nodes (blocks) of a region quadtree on $\mathbf{U}$,

K(L) is the key of L produced by s,
S(L) is the size, or alternatively the level, of L, and
C(L) is the color of the pixels intersecting L.

[1]This research was supported in part by Grant NSERC A7634.

In order to structure the file **F** using an adaptive cell method, the study area is partitioned into a set of blocks and/or subblocks which are defined in the following.

Definition 3. A *block* of *depth* d, $0 \le d \le maxd \le 2n$, where *maxd* is the predefined maximum depth of a block partition, is a rectangular region in the study area with a standard shape and a standard location that are the same as those of a region produced by recursively halving the study area d times with lines alternately perpendicular to the x and y axes. A block with its depth equal to *maxd* is called a *minimal block.*

Definition 4. A *subblock* of *depth d*, where $maxd < d \le 2n$, is a rectangular region in the study area with a shape and a location that are the same as those of a region produced by recursively halving the study area d times with lines alternately perpendicular to the x and y axes. Within each *minimal block* there exist at most two different depths of subblocks, i.e., d' and d'' such that $d'' = d' + 1$.

Throughout this paper, *maxd* will be used to denote the predefined maximum depth of a block partition. The following definition is useful for defining an adaptive cell method.

Definition 5. A *fixed data bucket* is a bucket which contains no more than a predefined number of records, b, and an *expandable data bucket* is a bucket which may contain more than b records by attaching one or more overflow fields to it.

An adaptive cell method is now defined that organizes the leaf nodes of a region quadtree into a file.

Definition 6. An *adaptive cell method* of organizing a file **F** is an abstract data type which:

(1) guarantees that, for every cell G1 and G2 and for every record L1 $\in$ **F** $\cap$ G1 and L2 $\in$ **F** $\cap$ G2, key(L1) < key(L2) if index(G1) < index(G2),

(2) guarantees that, for every subblock G1' and G2' of a minimal block G and for every record L1' $\in$ **F** $\cap$ G1' and L2' $\in$ **F** $\cap$ G2', key(L1') < key(L2') if index(G1')< index(G2'), and

(3) asserts that every block of depth d has exclusively one fixed data bucket associated with it if $d < maxd$; otherwise (the case of a minimal block), it has associated with it either a single fixed bucket exclusively or two or more expandable buckets that are contiguously located in physical memory such that:

a) each expandable bucket is exclusively associated with exactly one subblock of the minimal block, and

b) the overal *load factor*, i.e., the ratio of the number of existing records to the number of slots available, of these expandable buckets is within some predefined range.

It is understood from Definition 6 that every data bucket is associated with a block or a subblock in the study area. Consequently, each data bucket has associated with it a depth which is equal to the depth of the block or the subblock it corresponds to. For an illustration of the concept of Definition 6, consider the polygon network in Fig. 1. Let the predefined maximum depth of a block partition *maxd* = 4 (*maxd* is normally small so that the directory may be stored in main memory), the capacity of a data bucket $b = 5$, the capacity of an overflow field $b' = 3$, and the lower and upper limits of the load factor be 0.40 and 0.75, respectively. Then, for the image of Fig. 1a, the proposed scheme produces a partition as shown in Fig. 1b.

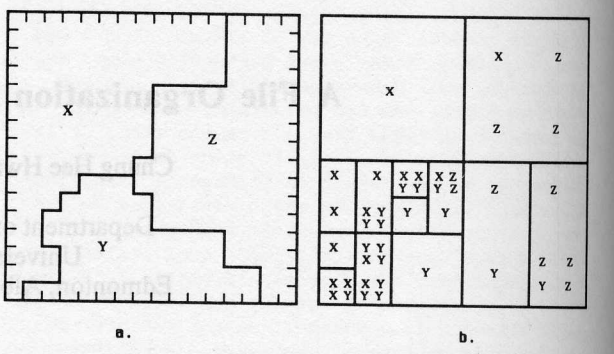


Fig. 1. Image of a Polygon Network and Partition of the Data Space.

3. Mapping between Regions and Data Buckets

Ordinarily the set of records in **F** is distributed over a number of data buckets, and each data bucket has associated with it a block or a subblock in the study area. The mapping between blocks and data buckets is achieved by a directory. A directory is a set of elements, each of which corresponds to a cell of size 2^{2n-d}, where d is the maximum of the depths of the existing blocks. Thus, a directory has associated with it a depth whose value is the same as d.

Each element of a directory has a pointer to a data bucket or a set of buckets which contains records describing the quadtree leaf nodes that intersect the corresponding cell in the study area. At depth d of a directory, there are altogether 2^d pointers, indexed from 0 to 2^d-1, which are not necessarily unique. The pointers of a directory are indexed in such a manner that a data bucket or a set of data buckets pointed to by a pointer with an index i contains all the records whose keys are prefixed with bits that are identical to the binary representation of i. That is, a data bucket or a set of data buckets pointed to by pointer 0 contains all the keys that start with d consecutive "0" bits, a data bucket pointed to by pointer 1 contains all the keys that start with d - 1 consecutive "0" bits followed by a "1" bit, and so on. Thus, the pointer i is guaranteed to find all the keys whose first d bits agree with the binary representation of i.

This indexing scheme is in fact equivalent to a Morton sequence [7] and naturally satisfies the first and second requirements of Definition 6. Fig. 2 illustrates the correspondence between the regions in the study area and directory elements (indexes are shown both in decimal and binary). Note that when the depth of a directory is odd, each pair of buddies at the deepest level are numbered consecutively from left to right, e.g., the pair 0 and 1 and the pair 2 and 3 in Fig. 2a.

[2] 010	[3] 011	[6] 110	[7] 111
[0] 000	[1] 001	[4] 100	[5] 101

a. Depth 3

[5] 0101	[7] 0111	[13] 1101	[15] 1111
[4] 0100	[6] 0110	[12] 1100	[14] 1110
[1] 0001	[3] 0011	[9] 1001	[11] 1011
[0] 0000	[2] 0010	[8] 1000	[10] 1010

b. Depth 4

Fig. 2. Correspondence Between Regions & Directory Elements

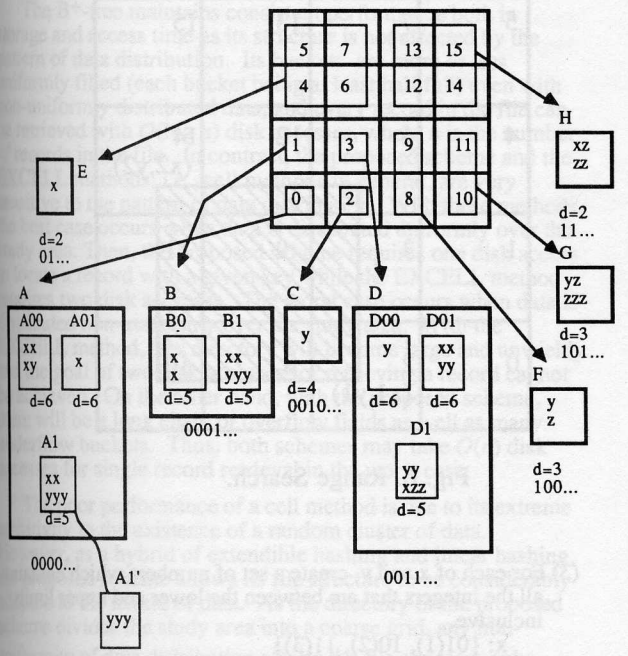


Fig. 3. Correspondence Between Blocks & Data Buckets.

The mapping between directory elements and data buckets, or sets of data buckets, is many-to-one. Fig. 3 shows the directory configuration corresponding to the partition shown in Fig. 1b. In Fig. 3, buckets *C*, *E*, *F*, *G* and *H* are fixed buckets, and buckets *A00*, *A01*, *A1*, *B0*, *B1*, *D00*, *D01*, and *D1* are expandable buckets (bucket *A1* has an overflow field attached to it). Note that all the records contained in a data bucket of depth *d* have the same bit pattern in their first *d* bits. Thus, bucket *F*, whose depth is 3, consists of the records whose keys start with "100", while bucket *AOO* whose depth is 6 contains all the records whose keys start with "000000". Also, notice that buckets *A00*, *A01* and *A1* contain 13 records in total while their capacity is 18. Thus, the load factor of these three expandable buckets is 13/18 = 0.72, which is within the predefined range. The directory has 2^4 pointers because the largest of the depths of existing blocks is 4 which is the same as *maxd*. Note that pointer 0 points to a set of buckets (*A00*, *A01* and *A1*) which contain all the records that start with "0000". The correspondence between subblocks and expandable data buckets, however, is not shown in the directory. That is, the corresponding pointer in the directory points to the starting address of a set of buckets that are physically located together, but it does not specify the correspondence between each of the subblocks and data buckets.

How the mapping between subblocks and data buckets are achieved will now be shown. Fig. 4 shows examples of a subblock partition of a minimal block. The subblocks of a minimal block are indexed in a similar manner as the cells corresponding to directory elements are indexed. In fact, when all the subblocks are the same size, they are indexed in the exactly same manner. Examples are shown in Figs. 4a and 4c. However, when there exist two different sizes of subblocks, the larger subblocks have two candidates for their index. In that case, the smaller of the two is selected for the index of the subblock as in Figs. 4b and 4d. As a result, when subblocks are of different sizes, the indexes of subblocks are not continuous.

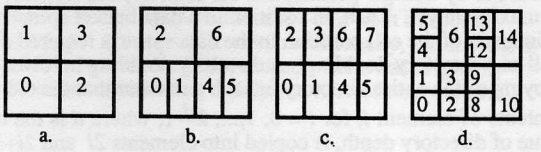
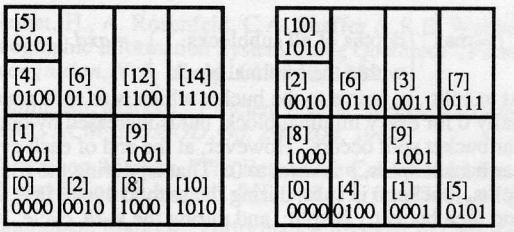


Fig. 4. Subblocks & Their Indexes.

Every subblock has an expandable data bucket associated with it. The keys of the records contained in a subblock of depth *d* have the same bit pattern in their leftmost *d* bit places. More explicitly, their leftmost *maxd* bits agree with the index of the minimal block the subblock belongs to, and the next (*d* - *maxd*) bits agree with the index of the subblock itself. Now, the mapping between subblocks and buckets is achieved by numbering each of the buckets that belongs to the same minimal block in a specific way as follows. Let a data bucket *D* be associated with a subblock whose index is *i*. Furthermore, let the maximum depth of existing subblocks be *d*. Then, *i* can be represented by a bit string *S* which is (*d* - *maxd*) bits long. Next, let *k* be a number represented by a bit string *S'* which is the reversed bit string of *S*. Then, *k* is the bucket number of *D*, i.e., *D* is the (*k* + *1*)*st* of the set of buckets that are contiguously located. The proof of this "reversed bit pattern" relation between *i* and *k* can be easily shown by induction (see [2] for a formal proof). Fig. 5 illustrates the correspondence between data buckets and subblocks.



a. Indexes of Subblocks. b. Data Bucket Numbers.

Fig. 5. Correspondence Between Subblocks & Data Buckets.

4. Dynamic Nature of the Scheme

The proposed file organization scheme allows a file structure to adapt its shape automatically to the nature of the data to be stored, i.e., the amount and the distribution pattern. The adaptability of the scheme is obtained mainly by a dynamic partition of the data space, which is implemented by *splitting* and *merging* mechanisms. In this section the merging mechansim is briefly described. See [3] for details of the dynamic file organization technique.

As more and more data is inserted in a file, data buckets overflow and this results in splitting of buckets. There are four kinds of splits possible. The first type of split occurs when a record is assigned to a data bucket that is full and pointed to by more than one pointer of the directory. In this case, the overflow bucket is split to resolve the collision, and the pointers in the directory are adjusted to reflect this split.

The second type of split arises when the overflow bucket is pointed to by a single pointer, and the directory has not reached its maximum yet. Then, in addition to a data bucket split, refinement of the cell partition in the data space is required as well as a directory doubling. A directory doubling involves copying of the entire directory in such a manner that the old contents of element i, for $i = 0, 1, ..., 2^d-1$, where d is the old value of directory depth, is copied into elements $2i$ and $2i+1$.

The third case occurs when the depth of the directory has reached its maximum already. Then, the overflow bucket is split into two expandable buckets, numbered 0 and 1, bucket 1 being physically allocated after bucket 0. This is called a *linear* bucket split.

The fourth type of split occurs when a record is assigned to an expandable bucket, and the load factor exceeds the upper limit as a result of insertion. Suppose a record is assigned to a data bucket of depth greater than d. Then, the record is first inserted into the bucket, or if necessary, into its overflow field, and the overall load factor of the set of buckets that are associated with the same minimal block is calculated and checked against the predefined range. If the load factor exceeds the upper limit, a *linear* bucket split is triggered, i.e., a new bucket is allocated at the end of the existing buckets of the set, and the bucket designated by the variable *next to split*, explained in the following, is split into two. If the load factor still exceeds the upper limit, the splitting process is repeated.

Similar to linear hashing [6], the following two variables are used to control linear bucket splits: j - *split level*, and p - *next to split*. The split level, j, indicates the level of linear splits within each minimal block. Initially, j is set to 0 for every minimal block but is increased as linear bucket splits are performed so that

$$j = \max \begin{pmatrix} \text{depths of all subblocks} \\ \text{within the minimal block} \end{pmatrix} - maxd.$$

Next to split , p, points to the bucket which is to split next. It is initially 0 for every minimal block, but is increased by one as a linear bucket split occurs. However, at the end of each cycle of linear bucket splits, p is reset to 0. That is, during the first cycle of splits, bucket 0 is split; during the second cycle, first, bucket 0, and then bucket 1 is split; and during the k–*th* cycle, buckets are split in the order of $1, ..., 2^{k-1}-1$.

5. Range Search

This section describes how the proposed file scheme supports range search. Given two points, (x_1,y_1) and (x_2,y_2), where $x_1 \le x_2$ and $y_1 \le y_2$, specifying a query rectangle, the proposed file scheme is able to retrieve every data bucket that contains the records describing the quadtree leaf nodes which overlap the query rectangle without retrieving any irrelevant data buckets. An algorithm for identifying the relevant data buckets for a given query rectangle is described using the example of Fig. 6. Suppose a directory has depth 4 which is the same as the predefined maximum depth. Suppose also that the shaded area in Fig. 6 is the query rectangle. Determination of the cells that intersect the query rectangle is done as follows:

(1) Using **Algorithm Access** in Appendix B, determine the indexes of cells in which (x_1,y_1) and (x_2,y_2) are contained. In this example, they are 2 and 14.

(2) Decompose the bit pattern of these indexes into their x and y-components. Let the x-component of the higher index be the upper limit of x. Similarly, determine the lower and upper limits of y.

Lower index:	0010 (2)	xlow:	01 (1)
	xyxy	ylow:	00 (0)
Higher index:	1110 (14)	xhigh:	11 (3)
	xyxy	yhigh:	10 (2)

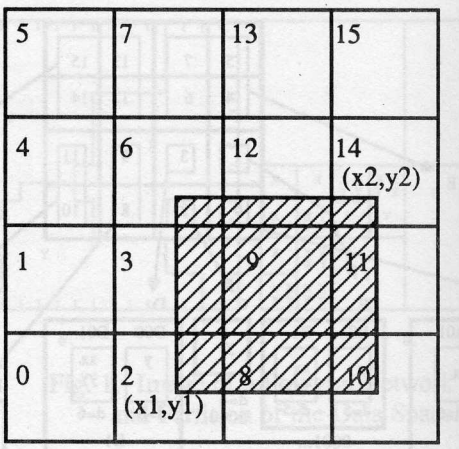


Fig. 6. Range Search.

(3) For each of x and y, create a set of numbers which contains all the integers that are between the lower and upper limits inclusive.
x: {01(1), 10(2), 11(3)}
y: {00(0), 01(1), 10(2)}

(4) Obtain a cross product of these sets, where each member of the resulting set is an integer produced by interleaving the x and y components. This set then indicates the cells that overlap the query rectangle. In this example, the following cells overlap the query rectangle: 2,3,6,8,9,10,11,12 and 14.

y \ x	01 (1)	10 (2)	11 (3)
00 (0)	0010 (2)	1000 (8)	1010 (10)
01 (1)	0011 (3)	1001 (9)	1011 (11)
10 (2)	0110 (6)	1100 (12)	1110 (14)

(5) Next, suppose some of the cells have been further subdivided, e.g., cells 9 and 12. As for cell 9, every expandable bucket associated with it should be retrieved since the cell is completely contained in the query rectangle. Retrieval of these buckets can be done using **Algorithm SequenRetrieve** in [3]. As for cell 12, the subblocks that intersect the query rectangle should be determined in a similar manner as the cells intersecting the query rectangle. The bucket number corresponding to each relevant subblock is then obtained by reversing the bits of the subblock index. The detailed description of the algorithm is given in **Algorithm RangeSearch** in Appendix C.

6. Performance of the Proposed Scheme

In this section, the performance of the proposed file organization scheme is compared with other file organization schemes in terms of access efficiency for single record retrieval. The object for comparison is a B^+-tree that has been proposed and implemented for representing a polygon network by a linear quadtree [1,8]. In addition, the EXCELL method is also used for comparison. Although the EXCELL method was originally used for representing a polygon network in vector format [9], the method is also useful for representing a polygon network by a quadtree.

The B+-tree maintains consistent performance both in storage and access time as its structure is not affected by the pattern of data distribution. Its buckets are more or less uniformly filled (each bucket being at least half full) even with non-uniformly distributed data, and every record in the file can be retrieved with $O(\log n)$ disk accesses, where n is the number of records in the file. In contrast, the proposed scheme and the EXCELL methods, i.e., cell methods in general, are very sensitive to the pattern of data distribution. With these methods, the best case occurs when data is distributed uniformly over the study area. Then, the proposed scheme requires one disk access to locate a record with a given key while the EXCELL method requires two disk accesses. The worst case occurs when data is distributed non-uniformly over the study area. With the EXCELL method, the directory will become large and unwieldy and the goal of two disk accesses for retrieving a record cannot be achieved. On the other hand, with the proposed scheme, there will be a long chain of overflow fields as well as many underflow buckets. Thus, both schemes may take $O(n)$ disk accesses for single record retrieval in the worst case.

The poor performance of a cell method is due to its extreme sensitivity to the existence of a random cluster of data. However, as a hybrid of extendible hashing and linear hashing, the proposed scheme allows its file structure to be considerably adapted to the nature of data. As the directory of the proposed scheme divides the study area into a coarse grid, any non-uniformity of data distribution affects the file structure only within a grid cell rather than over the entire study area. Furthermore, the probability of the worst case happening in practical data is expected to be exceedingly low. Since a worst case analysis does not provide meaningful conclusions, the performance of the proposed scheme has been simulated using a set of real data.

The scheme has been applied to a surficial geology map of the Wabamun area in Alberta, Canada (114°-115°W and 53.5°-54°N). Next, the same data have been used to estimate the performance of a B+-tree and the EXCELL method. It has been shown that the proposed scheme performs better than either a B+-tree or the EXCELL method in the expected number of disk accesses required to retrieve a record in a file, with a higher storage utilization ratio. See [3] for details. Although a formal proof cannot be given, it is conjectured that with the proposed scheme the expected number of disk accesses required for locating the record with a given key is constant irrespective of the file size, while that of B-trees or the (hierarchical) EXCELL method [11] grows logarithmically with the file size.

7. Conclusion

In most geometric databases, I/O operations are the bottleneck of their performance due to the large volume of data that should be handled. The proposed file organization scheme is an adaptive cell method which attempts to minimize the number of disk accesses in performing spatial queries. As a hybrid of interpolation hashing and extendible hashing, the proposed scheme combines the best features of both. First, the mapping between data buckets in physical storage and regions in the search space is interpolated rather than stored. Secondly, since a directory allows the search space to be divided into a coarse grid, any random cluster of data affects the file structure only within a grid cell rather than the entire file structure. Thirdly, a compromise between space and access time can be obtained by controlling the load factor.

Another important feature of the proposed scheme is that it handles spatial queries, range search in particular, efficiently by allowing a query to be decomposed into a set of subqueries within cell restrictions.

Experimental results with a set of real data show that the proposed scheme is superior to a B+-tree both in access efficiency and storage utilization. Additionally, the scheme is comparable to the EXCELL method which was originally proposed for representing a polygon network by vectors.

BIBLIOGRAPHY

1. Abel, D.J., "A B+-Tree Structure for Large Quadtrees", *Computer Vision, Graphics, and Image Processing*, Vol. 27, pp. 19-31, 1984.
2. Burkhard, Walter A., "Interpolation-Based Index Maintenance", *BIT*, Vol. 23, pp. 274-294, 1983.
3. Davis, Wayne A. & Chung Hee Hwang, "*File Organization Schemes for Geometric Data*", TR 85-14, Dept. of Computing Science, Univ. of Alberta, 1985.
4. Fagin, R., J. Nievergelt, N. Pippenger & H.R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM TODS*, Vol. 4, pp. 315-344, Sep. 1979.
5. Gargantini, Irene, "An Effective Way to Represent Quadtrees", *CACM*, Vol. 25, pp. 905-910, Dec. 1982.
6. Litwin,Witold, "Linear Hashing: A New Tool for File and Table Addressing" *Proc. 6th Int. Conf. Very Large Data Bases*, pp. 212-223, 1980.
7. Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures", *Comput. Surveys*, Vol. 16, pp. 187-260, Jun. 1984.
8. Samet, H., A. Rosenfeld, C.A. Shaffer & R.E. Webber, "A Geographic Information System Using Quadtrees", *Pattern Recognition*, Vol. 17, pp. 647-656, 1984.
9. Tamminen, Markku, "Efficient Spatial Access to a Data Base", *ACM-SIGMOD*, pp. 200-206, 1982.
10. Tamminen, Markku, "The Extendible Cell Method for Closest Point Problems", *BIT*, Vol. 22, pp. 27-41, 1982.
11. Tamminen, Markku, "Performance Analysis of Cell Based Geometric File Organizations", *Computer Vision, Graphics, and Image Processing*, Vol. 24, pp. 160-181, 1983.

APPENDIX

A. Data Structure

The file structure consists of a directory and data buckets. The directory has a *header* containing the depth of the directory followed by 2^d elements, where d is the depth of the directory. Each element of the directory is a 5-tuple of $<j, p, occ, over, ptr>$, where j is the split level, p is the bucket to be split next, *occ* is the number of records contained in the data bucket (or set of expandable buckets), *over* is the number of overflow fields employed, and *ptr* is the pointer to the data bucket (or set of expandable data buckets).

Each data bucket or overflow field contains a set of records, (K(L), S(L), C(L)). In addition to the set of records, each data bucket has a header that contains *db*, the bucket depth, and a pointer, *ptr*. If a bucket is an expandable one, *ptr* points to a chain of overflow fields attached to it; otherwise, it may be either ignored or used to point to the next bucket.

B. Algorithm Access

Input: file **F**, directory *dir* to **F**, and (x,y) ∈ **U**,
where **U** = $[0,2^n)^2$ is the study area.
Output: data bucket D which contains (K(L),S(L),C(L)) such that (x,y) ∈ L. (The record may be in an overflow field of D.)
Note: dir[i].A denotes field A of (i+1)*st* element of directory.

Step 1: *key* ← *s*(x,y)
Step 2: read *d*, depth of directory
Step 3: *i* ← $\lfloor (key / 2^{2n-d} \rfloor$ #determine dir index#
Step 4: *loc* ← *dir*[*i*].*ptr* #read pointer value#
Step 5: if *dir*[*i*].*j* = 0, goto Step 8
Step 6: #case of split level ≠0#
a. if *dir*[*i*].*p* = 0, #every bucket split#
1) #set subblock index with *j* bits of the *key*#
isub ← $\lfloor (key \bmod 2^{2n-d})/2^{2n-d-j} \rfloor$.
where *j* denotes *dir*[*i*].*j*
2) #calculate bucket no.#
bnum ← $\sum a_k 2^{m-1-k}$, for $0 \le k \le m-1$, where
isub is $\sum a_k 2^k$, for $0 \le k \le m-1$
b. otherwise
1) #set subblock index with (*j*-1) bits of the key#
isub ← $\lfloor (key \bmod 2^{2n-d})/2^{2n-d-j+1} \rfloor$,
where *j* denotes *dir*[*i*].*j*
2) #calculate bucket no.#
bnum ← $\sum a_k 2^{m-1-k}$, for $0 \le k \le m-1$, where
isub is $\sum a_k 2^k$, for $0 \le k \le m-1$
3) #if bucket split, adjust bucket no.#
if *bnum* < *dir*[*i*].*p* and (*d*+*j*)*th* bit of *key* = "1",
bnum ← *bnum* + 2^{j-1}
Step 7: #calculate address of the bucket#
loc ← *loc* + *bnum* * *unit-length*, where
unit-length is the bucket size
Step 8: access bucket D at *loc* and exit.

C. Algorithm RangeSearch

Input: file **F**, directory *dir* to **F**, and (x1,y1),(x2,y2) ∈ **U**,
where x1 ≤ x2 and y1 ≤ y2.
Output: retrieve every data bucket whose associated block or subblock in **U** intersects the query rectangle specified by (x1,y1) and (x2,y2).

Step 1: R ← { } #initialize index set#
Step 2: read *d*, depth of directory
Step 3: #calculate keys and indexes#
key1 ← *s*(x1,y1); *i1* ← $\lfloor key1 / 2^{2n-d} \rfloor$
key2 ← *s*(x2,y2); *i2* ← $\lfloor key2 / 2^{2n-d} \rfloor$
Step 4: if *i1* = *i2*, #trivial case#
R ← {*i1*} and goto Step 12
Step 5: #determine limits of index#
a. if *d* is even,
1) *xlow* ← $\sum a_{2k+1} 2^k$, for $0 \le k \le \lfloor m/2 \rfloor$
2) *xhigh* ← $\sum b_{2k+1} 2^k$, for $0 \le k \le \lfloor m/2 \rfloor$
3) *ylow* ← $\sum a_{2k} 2^k$, for $0 \le k \le \lceil m/2 \rceil$
4) *yhigh* ← $\sum b_{2k} 2^k$, for $0 \le k \le \lceil m/2 \rceil$
b. otherwise,
1) *xlow* ← $\sum a_{2k} 2^k$, for $0 \le k \le \lceil m/2 \rceil$
2) *xhigh* ← $\sum b_{2k} 2^k$, for $0 \le k \le \lceil m/2 \rceil$
3) *ylow* ← $\sum a_{2k+1} 2^k$, for $0 \le k \le \lfloor m/2 \rfloor$
4) *yhigh* ← $\sum b_{2k+1} 2^k$, for $0 \le k \le \lfloor m/2 \rfloor$
where *i1* (*i2*) is $\sum a_k 2^k$ ($\sum b_k 2^k$), for $0 \le k \le m-1$

Step 6: *ix* ← *xlow* #initialize *x* index#
Step 7: *iy* ← *ylow* #initialize *y* index#
Step 8: #compute index of relevant block#
if d is even, *i* ← **shuffle**(*ix*,*iy*);
otherwise, *i* ← **shuffle**(*iy*,*ix*), where
shuffle(V,W) = $\sum (2v_k + w_k) 2^{2k}$, for $0 \le k \le m-1$
given V = $\sum v_k 2^k$ and W = $\sum w_k 2^k$,
Step 9: R ← RU {*i*} #store the index#
Step 10: #continue until y upper limit is reached#
1) increase *iy* by 1
2) if *iy* ≤ *yhigh*, goto Step 8
Step 11: #continue until x upper limit is reached#
1) increase *ix* by 1
2) if *ix* ≤ *xhigh*, goto Step 7
Step 12: *loc* ← null #initialize#
Step 13: for each member *i* in R, perform
a. #determine address of bucket#
if *dir*[*i*].*ptr* ≠ *loc*, *loc* ← *dir*[*i*].*ptr*;
otherwise, goto Step 13.e
b. if *dir*[*i*].*j* = 0,
retrieve bucket at *loc* and goto Step 13.e
c. #linear splits have occurred#
if the block is totally contained in the query rectangle, retrieve every bucket belonging to the block using **Algorithm SequenRetrieve** and goto Step 13.e
d. #the block is partially contained#
1) R' ← { } #initialize subblock index set#
2) let (x1',y1') and (x2',y2'), where x1' ≤ x2' and y1' ≤ y2', be the points specifying the rectangle which is the intersection of the current minimal block and the query rectangle
3) compute R' in a similar manner to Steps 3-11
Note: in Step 8, *d* should be substituted with *j*
4) #compute the number of buckets#
if *dir*[*i*].*p* = 0, ***M*** ← 2^j;
otherwise, ***M*** ← $2^{j-1}+p$
5) for each member *isub* in R',
compute bucket number *bnum*;
if *bnum* < ***M***,
calculate address of bucket and retrieve
e. continue.