The Visual Computer

A comparison of the space requirements of multi-dimensional quadtree-based file structures*

Hanan Samet[1] and Robert E. Webber[2]

[1] Computer Science Department, The University of Maryland, College Park, MD 20742-3251, USA
[2] Department of Computer Science, Busch Campus, Rutgers University, New Brunswick, NJ 08903, USA

A comparison is made of the space requirements of pointer and a number of pointer-less implementations of multidimensional quadtree-based file structures. The database is assumed to be static. In order to make the comparison realistic, considerations such as computer byte sizes are taken into account, and fields are constrained to start on bit and byte boundaries where appropriate. In many practical cases, the pointer quadtree requires less space than the pointer-less quadtree. This effect is more pronounced for octrees and data of higher dimension. Empirical data from a cartographic batabase are used to support the analysis.

Key words: Image databases – Quadtrees – Octrees – Image compression

* The support of the National Science Foundation under Grant IRI8802457 is gratefully acknowledged

1 Introduction

Hierarchical data structures such as the quadtree (Klinger 1971) have proven to be a useful approach to organizing information present in image databases for use in applications in geographic information systems, image databases, and computer graphics (Samet 1984, 1990a, b). In this study we are interested in comparing the storage requirements of a number of different quadtree representations in a static environment. In particular, we present a detailed analysis of the pointer and pointer-less representations in the context of a realistic model of computer memory.

The rest of this paper is organized as follows. Section 2 contains a brief review of related quadtree data structures. Section 3 presents an overview of the different quadtree representations that we consider, while Section 4 contains the analysis of their storage requirements. The analysis is interpreted further in Section 5. Our approach is a general one and is applicable to quadtrees that represent images of arbitrary dimensionality and resolution. Our goal is to determine the maximum number of quadtree nodes that can be stored in a fixed amount of storage for an image of a given dimension and resolution (i.e., width). This is achieved by computing the cutoff values, in terms of the dimensionality and resolution of the image, at which the number of nodes that can be stored using the pointer representation requires less space than a number of different implementations of the pointer-less quadtree representation. These results are interpreted by examining the actual cuttoff values for images of different resolutions in two, three and four dimensions.

2 Background

Quadtrees have been used to represent a number of different types of spatial data. As an example, consider the quadtree approach to two-dimensional region data as illustrated by Fig. 1. It is based on the successive subdivision of the space containing a given region into four equal-size quadrants until homogeneous blocks (i.e., BLACK or WHITE for a binary image) are encountered. Figure 1b is the block decomposition of the region in Fig. 1a. This process is represented by a tree of degree 4 (i.e., each non-leaf node has four sons). The root node corresponds to the entire space containing the region. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the area represented by that node. The leaf nodes

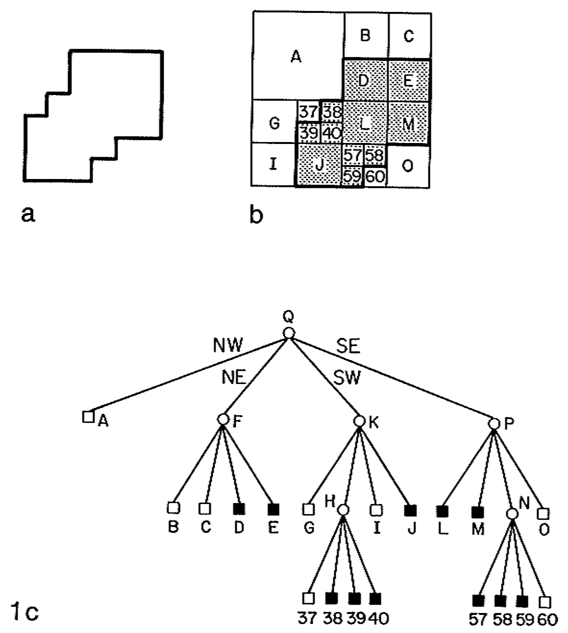


Fig. 1a–c. A region, its binary array, its maximal blocks, and the corresponding quadtree. **a** Region. **b** Block decomposition of the region in (**a**); blocks in the region are shaded. **c** Quadtree representation of the blocks in (**b**)

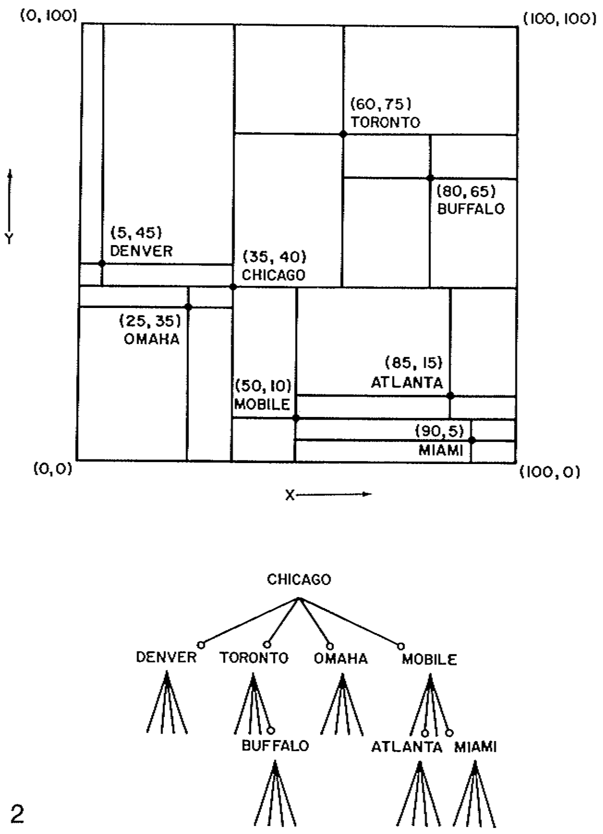
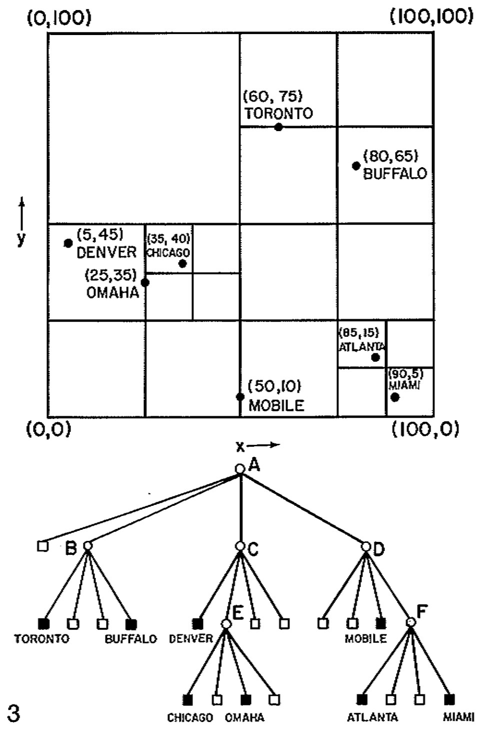
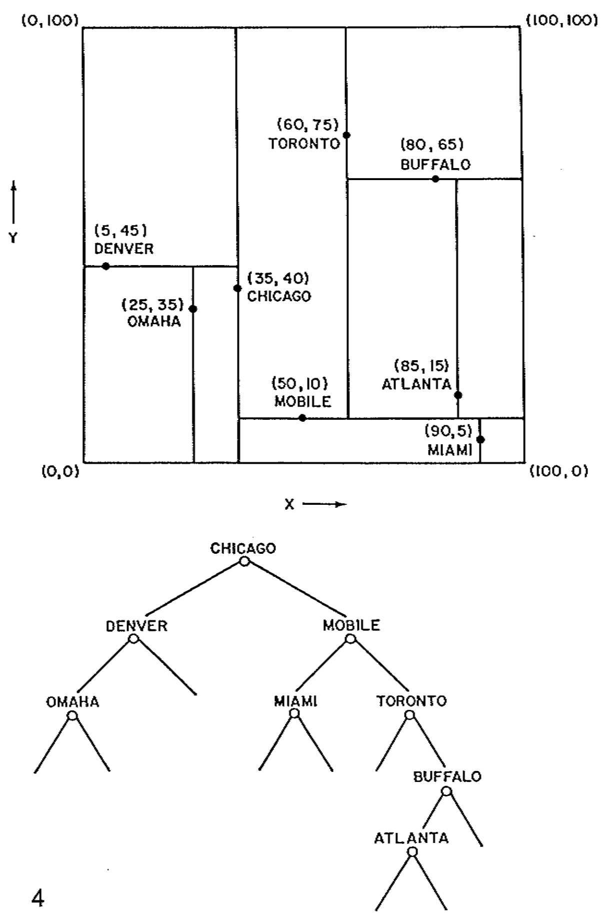


Fig. 2. A point quadtree and the records it represents

of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be BLACK or WHITE, depending on whether its corresponding block is entirely inside or entirely outside of the represented region. A leaf node at the maximum depth of the tree is called a *pixel*. All non-leaf nodes are said to be GRAY. The quadtree representation for Fig. 1b is shown in Fig. 1c.

Our discussion and examples of the quadtree are in the context of region data. Quadtrees and related data structures have been frequently studied in applications involving multidimensional point data. Most of these studies involved use of a variant known as the point quadtree (Finkel and Bentley 1974) and the k-d tree (Bentley 1975) for searching tasks. The difference between the region quadtree and these representations is that the region quadtree makes use of regular decomposition, whereas in the point quadtree (and the k-d tree) the actual data are used to guide the decomposition process. For example, Fig. 2 is an example of a point quadtree for a set of cities, while Fig. 3 is its corresponding region quadtree (termed a PR quadtree). Notice that the shape of the point quadtree depends on the order in which the cities are inserted, whereas this is not the case for the PR quadtree. The k-d tree (e.g., Fig. 4) is similar to the point quadtree with the modification that decomposition alternates between the x and the y coordinates (i.e., each node has out degree 2). A region k-d tree has also been defined and is termed a bintree (Knowlton 1980; Tamminen 1984; Samet and Tamminen 1988). Beckley et al. (1985) reported on an empirical study comparing point quadtrees and k-d trees. They concluded that neither representation is best for all queries. Matsuyama et al. (1984) also compared other variants of k-d trees and PR quadtrees. Other approaches include B-tree (Comer 1979) variants such as the R-tree (Guttman 1984; Roussopoulos and Leifker 1985; Faloutsos et al. 1987) and the k-d-B-tree (Robinson 1981). However, they





3

Fig. 3. A PR quadtree and the records it represents

Fig. 4. A K-d tree and the records it represents

4

are not of interest to us, since they often result in non-disjoint partitions of space.

In this paper our discussion and examples of the quadtree are in the context of region data and regular decomposition quadtrees. Thus we do not deal with point quadtrees and k-d trees. Note that since our comparison of pointer and pointer-less quadtree representations is only concerned with the number of blocks induced by the quadtree decomposition, our results are equally applicable to points, lines, faces, and other data for which the quadtree or octree can be viewed as providing a spatial index. In all of these cases, the quadtree is used to decompose the data until some other criterion of simplicity is satisfied [e.g., one point per quadrant (Orenstein 1982; Nelson and Samet 1987), one line segment per quadrant (Samet and Webber 1985), or one face per octant (Ayala et al. 1985; Carlbom et al. 1985; Fujimura and Kunii 1985)]. For a description of the use of quadtrees in a geographic database that includes points, lines, and regions, see (Samet et al. 1984, 1987).

3 Quadtree representations

One of the prime motivations for the development of the quadtree, and furthermore for different quadtree representations, has been a desire to save space. The conventional quadtree implementation is as a tree structure consisting of two types of nodes. Non-terminal nodes contain four pointers corresponding to the four subtrees. Terminal nodes contain four null pointers corresponding to the empty subtrees. In the rest of this paper we will be measuring the storage requirements in terms of the number of leaf nodes, which will be denoted by L. It is well known that the total number of nodes in a quadtree is $(4/3)\cdot(L-1)+1$ (Knuth 1973). Each node requires space for four pointers where each pointer can be encoded with approximately $\log(4/3)\cdot L)$ bits. Thus the total storage necessary for such a pointer representation is approximately $(16/3)\cdot L\cdot\log(4/3\cdot L)$ bits. Note that all logarithms in this paper are with respect to base 2.

The pointer representation seems verbose and as a result there has been a considerable amount of interest in pointer-less quadtree representations. Pointer-less representations can be grouped into two categories. The first represents the quadtree as a traversal of its constituent nodes (Kawaguchi and Endo 1980). For example, letting 'B', 'W', and 'G' correspond to BLACK, WHITE, and GRAY nodes respectively, and assuming a traversal in the order NW, NE, SW, and SE, then the quadtree of Fig. 1 would be represented by

GWGWWBBGWGWBBBWBGBBGBBBWW.

This approach requires two bits to represent each node in the quadtree. Therefore, a quadtree with L leaf nodes requires approximately $2\cdot(4/3)\cdot L$ bits.

The second approach treats the quadtree as a collection of the leaf nodes comprising it. Each node is represented by a pair of numbers (Gargantini 1982c). The first number is the depth (also referred to as level) of the tree at which the node is located. Assuming that the root is at level 0, and that the maximum level value is h, then we need $\log(h)$ bits to represent the level of a node. The second number is termed a *locational code*. There are a number of choices for the second number. It can be represented by either a variable or a fixed number of bits. Using a variable number of bits, one variant of the locational code is formed by a concatenation of base 4 digits corresponding to directional codes that locate the node along a path from the root of the quadtree. The directional codes take on the values 0, 1, 2, 3 corresponding to quadrants NW, NE, SW, SE, respectively. For example, the pair of numbers (3,312) are decoded as follows: 312 is the base 4 locational code and denotes a node at level 3 that is reached by a sequence of transitions, SE, NE, and SW, starting at the root. Using a fixed number of bits, the locational code is formed by interleaving the bits comprising the coordinate values of a specific point in the leaf node (e.g., the pixel in the lower left corner). A quadtree representation based on the use of locational codes is called *linear quadtree* by Gargantini (1982a, c) (because the addresses are keys in a linear list of nodes) and a *leafcode* by Oliver and Wiseman (1983).

For the second approach described above we have the following storage requirements. In each of its two variants, each node requires two bits per level of depth of the node plus a number of bits equal to the base 2 logarithm of the depth of the node in order to specify the level at which the node is found.

The traversal-based pointer-less approach (i.e., the first one discussed) is useful for operations that require each element in the tree to be visited in the same order as the traversal that serves as the basis of the representation. However, it is difficult to perform operations which require accessing elements of the quadtree at random. For example, consider the quadtree of Fig. 1. Suppose that we are given a pointer to node F which is the NE son of node Q. In order to locate the node which is the SE son of node Q (i.e., P), we must sequentially visit every node in the SW son of node Q (i.e., K). The variant of the second pointer-less approach that uses a variable number of bits to represent the locational code is somewhat cumbersome because nodes are not uniformly represented. Thus the variant that uses a fixed number of bits is more general, and in the rest of this paper we shall use it (and refer to it as a linear quadtree) in our comparison of the storage requirements of pointer and pointer-less quadtree representations. In particular, for a quadtree with L leaf nodes and a maximum level of h, the fixed variant requires a total of $L\cdot(2\cdot h+\log(h))$ bits.

4 Analysis

When comparing pointer and pointer-less quadtree representations, it is generally accepted that the pointer representation is more flexible for programming. On the other hand, pointer-less representations have been viewed as being more compact. This second property has been considered particularly important when working with spaces of dimensionality greater than two. In this section we show that for a given maximum depth of a quadtree and dimensionality of the space, there exists a cutoff value such that if the number of leaf nodes is less than the cutoff value, then the pointer quadtree representation is more compact than a pointer-less representation that makes use of a fixed number of bits.

In past analyses (Gargantini 1982a–c) the number of bits required for the locational code of the linear quadtree was allowed to vary with the maximum depth of the quadtree, whereas in the pointer quadtree the width of each pointer field was constrained to be the same regardless of the number of leaf nodes that needed to be distinguished. Moreover, the implementation of the pointer quadtree was needlessly verbose. For example, past analyses assume the existence of father links which, of course, is not necessary for the proper use of pointer quadtrees. In the following we examine a particularly compact method of implementing a pointer quadtree and compare its storage requirements with that of a linear quadtree. Our analysis ignores the data fields and concentrates on the number of bits necessary to represent the inherent tree structure of the quadtree. We also describe the effect of varying the alignment requirements of pointers in terms of bit and byte boundaries. We assume 8-bit bytes, although it should be clear that the same analysis can be applied to other byte sizes.

For a d-dimensional image containing L leaf nodes such that the maximum level of any node is h, the number of bits required by the linear quadtree is

$$L\cdot(d\cdot h+\log(h+1)). \tag{1}$$

On the other hand, the situation for the pointer quadtree is more complicated. A pointer quadtree has two types of nodes (internal and external). Internal nodes consist of 2^d pointers, whereas external nodes do not require pointers (see also Doctor and Torborg 1981; Meagher 1982b; Yau and Srihari 1983; Okawara et al. 1988). However, we do need to distinguish between the two node types. This requires one bit. We propose to add this bit as part of the pointer field that points at the node being described rather than in the node being described. This is analogous to a coding convention used in threading (Knuth 1973) that distinguishes between links and threads by stipulating that links can be detected by virtue of pointing to nodes with greater addresses, while threads point to nodes with lesser addresses. Thus, in our implementation no storage is attributed to the leaf nodes; instead, all the excess storage is accounted for in the internal nodes. The number of internal nodes is

$$(L-1)/(2^d-1)$$

which is bounded from above by

$$L/(2^{d-1}).$$

In our analysis we shall use this upper bound, as it makes the manipulation of the equations more tractable. It should be clear that the slight overestimation of the number of nodes in the pointer quadtree will not affect our final results except for small values of L. Thus at times our derived cutoff values will be lower than they would be had we not overestimated the number of internal nodes (and hence the total) in the quadtree. Interestingly, as we shall see in Sect. 5, this overestimation affects the cutoff value for the image in Fig. 1.

Note that each pointer field needs only to be wide enough to distinguish between all the possible nodes in a particular tree whose number is

$$L\cdot(1+1/(2^d-1))$$

or

$$L\cdot 2^d/(2^d-1).$$

Thus the total number of bits needed to store a pointer quadtree is

$$(L/(2^d-1))\cdot(2^d\cdot(1+\log(L\cdot 2^d/(2^d-1)))). \tag{2}$$

Using Eqs. (1) and (2) we see that in order for the linear quadtree to be more compact (i.e., require less bits) than the pointer quadtree, the following relation must hold:

$$L\cdot(d\cdot h+\log(h+1)) \\ <(L/(2^d-1))\cdot(2^d\cdot(1+\log(L\cdot 2^d/(2^d-1)))). \tag{3}$$

Factoring L out of Eq. (3) and letting $m=2^d$ and $n=2^d-1$ enables Eq. (3) to be rewritten as

$$d\cdot h+\log(h+1)<(m/n)\cdot(1+\log(L\cdot m/n)). \tag{4}$$

Observe that m/n is approximately 1. Solving (4) for L leads to

$$(n/m)\cdot 2^{(n/m)\cdot(d\cdot h+\log(h+1))-1}<L. \tag{5}$$

The relation given by Eq. (5) requires some interpretation. In particular, letting C denote the left side of Eq. (5), it indicates that as long as $C<L$, then the linear quadtree requires less bits than the pointer quadtree. In other words, for a given d and h, if the number of leaf nodes (i.e., L) is less than or equal to C (i.e., $L\leq C$), then the pointer quadtree requires a smaller or equal number of bits than the linear quadtree. Thus C is a cutoff value for the number of leaf nodes. We must also show that the cutoff value is unique. This is easy to see because C can be written as a function of d and h, which is monotonically increasing in both

d and h as long as each of them is greater than or equal to 2.

The cutoff value obtained in Eq. (5) is based on a continuous model in the sense that the derivatioin does not restrict the pointer, level, depth, and node type fields to lie on bit boundaries as would be required in an actual implementation. Letting $\lceil x \rceil$ represent the ceiling of x, we can rewrite Eq. (4) to incorporate a restriction that these fields comprise an integer number of bits by:

$$d \cdot h + \lceil \log(h+1) \rceil < (m/n) \cdot (1 + \lceil \log(L \cdot m/n) \rceil). \tag{6}$$

Rearranging the sides of Eq. (6) yields

$$(n/m) \cdot (d \cdot h + \lceil \log(h+1) \rceil) - 1 < \lceil \log(L \cdot m/n) \rceil. \tag{7}$$

By approximating $\log(L \cdot m/n)$ by $\log(L \cdot m/n) + 1$, the inequality given by Eq. (7) can be rewritten as

$$(n/m) \cdot (d \cdot h + \lceil \log(h+1) \rceil) - 1 - 1 < \log(L \cdot m/n). \tag{8}$$

Solving Eq. (8) for L results in

$$(n/m) \cdot 2^{(n/m) \cdot (d \cdot h + \lceil \log(h+1) \rceil - 2} < L \tag{9}$$

Unfortunately, bit addressability is awkward on most computer architectures. Therefore, typical implementations lead to a further restriction so that the encoding for a given node starts on a byte boundary. In this analysis, we shall assume 8-bit bytes. Let $\{x\}$ denote the quantity $8 \cdot \lceil x/8 \rceil$. In the following, we pack the pointers across byte boundaries while still requiring each pointer field to comprise an integer number of bits. Restricting each node to start on a byte boundary results in Eq. (4) being rewritten as

$$\{d \cdot h + \lceil \log(h+1) \rceil\}$$
$$< (1/n) \cdot \{m \cdot (1 + \lceil \log(L \cdot m/n) \rceil)\}. \tag{10}$$

Multiplying both sides of Eq. (10) by n yields

$$n \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} < \{m \cdot (1 + \lceil \log(L \cdot m/n) \rceil)\}.$$

Note that

$$\{m \cdot (1 + \lceil \log(L \cdot m/n) \rceil)\}$$
$$= 8 \cdot \lceil m \cdot (1 + \lceil \log(L \cdot m/n) \rceil)/8 \rceil$$

implies

$$\{m \cdot (1 + \lceil \log(L \cdot m/n) \rceil)\}$$
$$\leqq 8 \cdot (m \cdot (1 + \lceil \log(L \cdot m/n) \rceil)/8 + 1) \tag{11}$$

Applying the following transformations to Eq. (11) yields Eq. (12)

$$(n \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - 8)/m$$
$$< 1 + \lceil \log(L \cdot m/n) \rceil$$
$$(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - (8/m)$$
$$< 1 + \lceil \log(L \cdot m/n) \rceil$$
$$(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - (8/m) - 1$$
$$< \lceil \log(L \cdot m/n) \rceil$$
$$(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - (8/m) - 2$$
$$< \log(L \cdot m/n)$$
$$(n/m) \cdot 2^{(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - (8/m) - 2} < L. \tag{12}$$

At this point, it is interesting to compare the cutoff values of Eq. (9) and Eq. (12). Let us assume that the difference between $\{d \cdot h + \lceil \log(h+1) \rceil\}$ and $d \cdot h + \lceil \log(h+1) \rceil$ can be ignored, i.e., for the sake of this comparison, we waive the requirement that each node of the linear quadtree start on a byte boundary while still requiring each node of the pointer quadtree to start on a byte boundary. Thus no bits are wasted in the case of the linear quadtree, while they are still wasted in the case of the pointer quadtree. In this case, the cutoff value when a node must start on a byte boundary is $2^{-8/m}$ times the integer field cutoff value. Observe that in the two-dimensional case, $2^{-8/m}$ is 1/4, in the three-dimensional case it is 1/2, and in the four-dimensional case it is about 1/1.4.

In some applications it may be inconvenient to unpack the bytes containing the four pointers prior to accessing a particular son. In such a case, the pointer quadtree is further restricted so that each pointer starts on a byte boundary. Incorporating this restriction, and using the same notation as before, Eq. (4) is rewritten as

$$\{d \cdot h + \lceil \log(h+1) \rceil\} < (m/n) \cdot (1 + \{\log(L \cdot m/n)\}). \tag{13}$$

Rearranging the sides of Eq. (13) and solving for L yields

$$(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - 1 < \{\log(L \cdot m/n)\}$$
$$(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil\} - 1 - 8 < \log(L \cdot m/n)$$
$$(n/m) \cdot 2^{(n/m) \cdot \{d \cdot h + \lceil \log(h+1) \rceil - 1 - 8} < L. \tag{14}$$

The cutoff value of Eq. (14) can be interpreted by examining the relationship between $\{d \cdot h + \lceil \log(h+1) \rceil\}$ and $d \cdot h + \lceil \log(h+1) \rceil$, and using the cutoff value of Eq. (9) as a point of reference. The best case of the linear quadtree arises when we ignore the difference between $\{d \cdot h + \lceil \log(h+1) \rceil\}$ and $d \cdot h + \lceil \log(h+1) \rceil$, i.e., for the sake of this comparison, we waive the requirement that each node of the linear quadtree start on a byte boundary while still requiring each pointer of the pointer quadtree to start on a byte boundary. In this case the cutoff value when each pointer must start on a byte boundary is 2^{-7} times the integer field cutoff value. Thus the cutoff value beyond which the linear quadtree would be more compact than the pointer quadtree is significantly lower. On the other hand, the worst case of the linear quadtree arises when the difference between $\{d \cdot h + \lceil \log(h+1) \rceil\}$ and $d \cdot h + \lceil \log(h+1) \rceil$ is a maximum. For a byte size of 8, the maximum difference between $\{x\}$ and x is 7 when x is an integer. Now, as n/m approaches 1, the integer field cutoff value of Eq. (9) and the each pointer on a byte boundary cutoff value of Eq. (14) would tend to be the same. In other words, the cutoff values are the same when the configuration of nodes is such that the linear quadtree wastes 7/8ths of one byte for each locational code.

5 Empirical interpretation

In order to gain a better understanding of the interrelationships between the different cutoff values, we have tabulated them in Table 1[1] for two-dimensional images whose sizes range from $2^3 \times 2^3$ to $2^{15} \times 2^{15}$. The table assumes 8-bit bytes. The maximum size of these images is much larger than what is currently used in practical applications. For each image size we have also indicated the maximum number of leaf nodes in the corresponding quadtree.

As a simple example, consider the quadtree in Fig. 1. When all fields are restricted to lie on integer bit boundaries, the table indicates that when the quadtree has more than 12 leaf nodes the linear quadtree is more efficient spacewise than the pointer quadtree. However, in this example it turns out that the pointer quadtree (150 bits) is more efficient than the linear quadtree (152 bits). In fact, the true cutoff value for depth 3 is really somewhere between 22 and 25 (there cannot be a quadtree with 23 or 24 leaf nodes since the number of leaf nodes modulo 3 is always 1). The reason for the discrepancy is easy to see by recalling the derivation of Eq. (2), whose consequence is that all of our cutoff values are really lower bounds, since they are based on an overestimation of the total number of internal nodes in the quadtree. The effect of this overestimation on the true value of the cutoff becomes insignificant as the depth of the quadtree is increased because for small values of L, the pointer quadtree is always superior to the linear quadtree when the depth is sufficiently large.

From a practial standpoint, the most realistic of the restrictions on the alignment of the fields and pointer values is the one that forces a node to start on a byte boundary while at the same time forcing

[1] The cutoff values of relations (9), (12), and (14) are somewhat lower than the actual values indicated by Eqs. (6), (10), and (13), respectively, because of the approximation of $\log(L \cdot m/n)$ by $\log(L \cdot m/n)+1$, as well as roundoff errors in the computation of natural logarithms and non-integer powers of two. However, as the depth increases, these differences become insignificant

Table 1. Minimum leaf node counts for locational codes being better than pointers

Depth	Continuous: relation (5)	Each field on bit boundary: relation (9)	Each node on byte boundary: relation (12)	Each pointer on byte boundary: relation (14)	Maximum number of nodes
3	19	12	3	0	64
4	68	34	192	6	256
5	227	161	192	6	1024
6	736	457	192	6	4096
7	2337	1292	12288	384	16384
8	7306	3653	12288	384	65536
9	22574	17378	12288	384	262144
10	69101	49152	12288	384	1048576
11	209928	139023	786432	24576	4194304
12	633807	393216	786432	24576	16777216
13	1903591	1112183	786432	24576	67108864
14	5691899	3145728	786432	24576	268435456
15	16954095	8897462	50331648	1572864	1073741824

all fields to be of integer length (measured in bits). It results in little wasted space because of the multiplicative effect of the 2^d pointers per node. In particular, for a byte size of b, there is no wasted space whenever $2^d \bmod b = 0$, which is true whenever $d \geqq 3$ and $b = 8$. In two dimensions we waste at most 4 bits when $b = 8$.

In fact, we see that for some depth values, the storage requirements of the pointer quadtree are never worse than the linear quadtree. This effect is very clear for images of greater than two dimensions as shown in Table 2 and discussed below. The pointer quadtree is at its worst in comparison with the linear quadtree when both nodes and their individual constituent pointers must start on byte boundaries.

Figure 5 is an example of a complicated image which is a map from a geographic information system that uses quadtrees (Samet et al. 1984). It is a map of a floodplain region with 5266 leaf nodes and is of depth 9. Notice that when nodes are restricted to start on a byte boundary, the cutoff value at depth 9 is 12,288. Under this restriction, the pointer quadtree of Fig. 5 requires 12,285 8-bit bytes while the linear quadtree requires 15,798 8-bit bytes. Thus Fig. 5 is more compactly encoded by the pointer quadtree.

Table 2 contains a more thorough tabulation of the effect of restricting all nodes to start on byte boundaries by varying the dimension of the image as well as its depth. Again, we are using images whose depth ranges from 3 to 15 with a byte size of 8 bits. The images are either of two, three, or four dimensions. For each dimension and depth, the table contains the value of the difference between the log of the maximum number of nodes in a quadtree of this depth and dimension and the log of the cutoff value for this restriction. In three dimensions, for depths of 3, 5, 8, 10, and 15, the storage requirements of the pointer quadtree are never worse than the linear quadtree. Similarly, in four dimensions for trees of depth 3, 4, 6, 8, 10, 12, and 14, the same result holds. In fact, in the three and four-dimensional cases the cutoff values are so close to the maximum node counts, that in all practical cases the pointer quadtree would still occupy less space than the linear quadtree.

It is important to realize that at the cutoff value, both the linear and pointer quadtrees are equally compact. An interesting question is how quickly does the advantage of the linear quadtree grow as we move above and beyond the cutoff value. Returning to the original continuous cutoff value of Eq. (5), we note that in order for the linear quadtree to save k bits per leaf node, the following relation must hold:

$$L \cdot (d \cdot h + \log(h+1)) + k \cdot L < (L/n) \cdot (m \cdot (1 + \log(L \cdot m/n))). \quad (15)$$

Table 2. The log of the maximum number of leaf nodes minus log of cutoff

Depth	2D	3D	4D
3	4.42	−1.81	−0.41
4	0.42	1.19	−3.91
5	2.42	−2.81	0.09
6	4.42	0.19	−3.41
7	0.42	3.19	0.59
8	2.42	−0.81	−2.91
9	4.42	2.19	1.59
10	6.42	−1.81	−2.41
11	2.42	1.19	1.59
12	4.42	4.19	−1.91
13	6.42	0.19	2.09
14	8.42	3.19	−1.41
15	4.42	−0.81	2.09

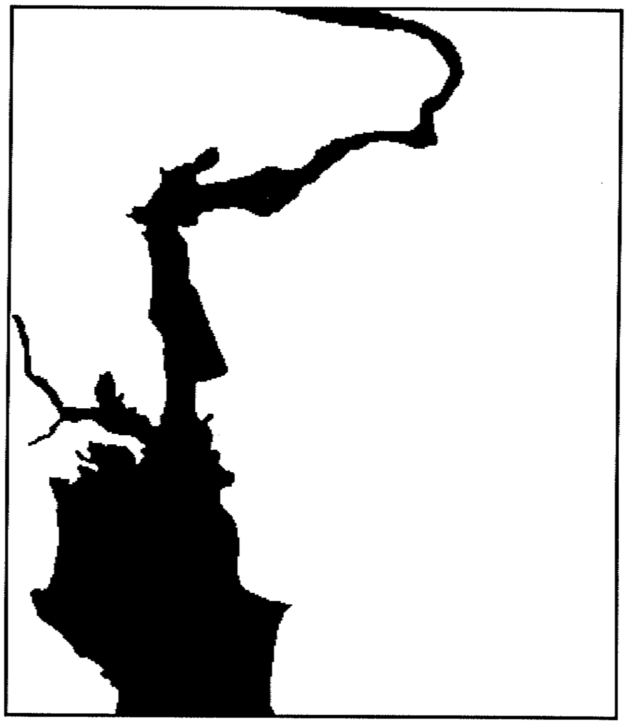


Fig. 5. Example map of a floodplain

Rearranging Eq. (15) in a manner analogous to that used in the previous formulas and solving for L yields

$$(n/m)\cdot 2^{(n/m)\cdot(d\cdot h+\log(h+1)+k)-1}<L. \tag{16}$$

From Eq. (16) we see that in order to save k bits per node, the value of L must be increased by a multiplicative factor of $2^{(n/m)\cdot k}$ (recall that n/m is $1-2^{-d}$). As an example, for a two-dimensional image, in order for the linear quadtree to save more than one bit per node over the pointer quadtree, we need to increase the number of leaf nodes by approximately 68% since $2^{(n/m)\cdot k}$ is 1.68.

If we knew that all the leaf nodes were stored at higher memory locations than the internal nodes, then we would not need the flag bit of Eq. (2) to distinguish between pointers to leaf and internal nodes. For this case, the number of bits being saved per leaf node (which we denoted by k) is m/n, i.e., the ratio of the total number of nodes to leaf nodes. This savings in the pointer representation would raise the cutoff value in the continuous case (as well as all the other cases) by a factor of $2^{(n/m)\cdot(m/n)}$, which is 2.

6 Concluding remarks

We have shown that the storage requirements of the pointer quadtree are significantly reduced, and for typical data the pointer quadtree often requires less space than the linear quadtree, when we make use of the following properties:

1. Pointers can be packed
2. Pointers need not be larger than is necessary to distinguish between the nodes in the tree
3. NIL pointers need not be stored explicitly
4. Father links need not be stored

This is especially true for three-dimensional (e.g., octrees (Hunter 1978; Meagher 1982a; Jackins and Tanimoto 1980)) and higher dimensional data (Samet and Tamminen 1985).

Of course, as nodes are inserted and deleted from the quadtree, the storage requirements of the pointer quadtree will change more abruptly than those of the linear quadtree. This is because each additional node in the linear quadtree requires a fixed amount of extra storage (i.e., $d\cdot h+\lceil\log(h+1)\rceil$ bits per node in a d dimensional image of height h), whereas the space required for each node in the pointer quadtree is a function of the number of nodes in the tree, which can change dynamically. On the other hand, additional nodes may result in an increase in the number of bits required for a pointer, resulting in a need to reallocate memory. Thus our comparison is really restricted to files of static size. From an implementation point of view, it makes more sense to interpret our comparison as addressing the issue of which representation is most likely to permit the storage of the largest quadtree of a given resolution in a fixed amount of memory (e.g., the size of a disk or available amount of core).

Gargantini (1982a) points out that for a binary image represented as a linear quadtree we only need to store the BLACK nodes; the WHITE nodes can be inferred by use of a procedure that simulates the quadtree construction process (see also two-dimensional run-encoding (Lauzon et al. 1985)). The potential space savings resulting from the use of such a technique depend on the probability of a leaf node being BLACK. Of course, the same probabilistic information could also be used to compress the pointer representation by reducing the number of nodes through the use of techniques such as the FBW sequence of approximations (Samet 1985) which are based on forests (Jones and Iyengar 1984). A comparison of the merit of the application of such approaches to the linear and pointer quadtree representations is beyond the scope of this paper.

It should be borne in mind that in this paper we have only focussed on traditional quadtree-like data structures (i.e., a branching factor of 2^d for a d-dimensional image). The bintree (Knowlton 1980; Tamminen 1984; Samet and Tamminen 1988) is an alternative data structure that has a branching factor of 2 at each node regardless of the dimension of the image. A linear bintree (Tamminen 1984) can be defined in a manner analogous to the linear quadtree. In this case, performing the same comparison as made in this paper between the linear bintree and the pointer bintree would yield the opposite result. In particular, it is easy to see that the linear bintree is almost always more efficient spacewise than the pointer bintree. For example, applying the analysis of the continuous model, as illustrated by the derivation of Eq. (3) in Sect. 4, to a bintree with L' leaf nodes, we find that the following relation must hold in order for the linear bintree to be more compact than the pointer bintree:

$$L'\cdot(d\cdot h+\log(h+1))<2\cdot L'\cdot(1+\log(2\cdot L')). \tag{17}$$

Applying transformations to Eq. (17) similar to those used earlier we find that the linear bintree will require less bits than the pointer bintree as long as

$$\sqrt{(h+1)\cdot 2^{d\cdot h-4}} < L.$$

Observing that $2^{d\cdot h}$ is the maximum number of leaves in a bintree of dimension d and depth $d\cdot h$, it should be apparent that this threshold is so low that the number of leaf nodes in most images will fall above it. An alternative justification for this result is that in quadtree-like data structures, as the dimension of the space increases, the majority of the nodes are leaf nodes while the proportion of internal nodes is considerably smaller. On the other hand, for the bintree the number of internal nodes is always one less than the number of leaf nodes regardless of the dimension of the space. Thus, the fact that for quadtrees the proportion of internal nodes is lower means that the linear quadtree requires proportionally more space than the linear bintree when compared to their pointer counterparts. It should be noted that although the pointer quadtree is often more compact than the linear quadtree, and the linear bintree is usually more compact than the pointer bintree, we do not yet have a data model that will allow us to determine on the average which of the linear bintree or the pointer quadtree is more compact.

Our discussion of the quadtree has been in the context of region data. However, the comparison of this paper is equally applicable to points, lines, face, and other data for which the quadtree or octree can be viewed as providing a spatial index. In these cases, the quadtree is used to decompose the data until some other criterion of simplicity is satisfied (e.g., one point per quadrant (Orenstein 1982; Nelson and Samet 1987), one line segment per quadrant (Samet and Webber 1985), or one face per octant (Ayala et al. 1985; Carlbom et al. 1985; Fujimura and Kunii 1985)). A description of the use of linear quadtrees for points, lines, and regions can be found in Samet et al. 1984.

Acknowledgments. We would like to thank Timos Sellis for his comments and Jim Purtilo for help in producing Tables 1 and 2.

References

Ayala D, Brunet P, Juan R, Navazo I (1985) Object respresentation by means of nonminimal division quadtrees and octrees. ACM Trans Graph 4(1):41–59

Beckley DA, Evens MW, Raman VK (1985) Multikey retrieval from k-d trees and quad-trees. Proc SIGMOD Conf, Austin, Texas (May 1985), pp 291–301

Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517

Carlbom I, Chakravarty I, Vanderschel D (1985) A hierarchical data structure for representing the spatial decomposition of 3-D objects. IEEE Comput Graph Appl 5(4):24–31

Comer D (1979) The Ubiquitous B-tree. ACM Comput Surv 11(2):121–137

Doctor LJ, Torborg JG (1981) Display techniques for octree-encoded objects. IEEE Comput Graph Appl 1(1):29–38

Faloutsos C, Sellis T, Roussopoulos N (1987) Analysis of object oriented spatial access methods. Proc SIGMOD Conf, San Francisco (May 1987), pp 426–439

Finkel RA, Bentley JL (1974) Quad trees: a data structure for retrieval on composite keys. Acta Inf 4(1):1–9

Fujimura K, Kunii TL (1985) A hierarchical space indexing method. Proc Comput Graph '85 (Tokyo 1985), T1–4, pp 1–14

Gargantini I (1982a) An effective way to represent quadtrees. Commun ACM 25(12):905–910

Gargantini I (1982b) Linear octtrees for fast processing of three dimensional objects. Comput Graph Image Proc 20(4):365–374

Gargantini I (1982c) Detection of connectivity for regions represented by linear quadtrees. Comput Mathe Appl 8(4):319–327

Guttman A (1984) R-trees: a dynamic index structure for spatial searching. Proc SIGMOD Conf, Boston (June 1984), pp 47–57

Hunter GM (1978) Efficient computation and data structures for graphics. PhD Dissertation, Dep Electrical Eng Comput Sci, Princeton Univ, Princeton NJ

Jackins CL, Tanimoto SL (1980) Oct-trees and their use in representing three-dimensional objects. Comput Graph Image Proc 14(3):249–270

Jones L, Iyengar SS (1984) Space and time efficient virtual quadtrees. IEEE Trans Pattern Anal Mach Intell 6(2):244–247

Kawaguchi E, Endo T (1980) On a method of binary picture representation and its application to data compression. IEEE Trans Pattern Anal Mach Intell 2(1):27–35

Klinger A (1971) Patterns and search statistics. In: Rustagi JS (ed) Optimizing Methods in Statistics. Academic Press, New York, pp 303–337

Knowlton K (1980) Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes. Proc IEEE 68(7):885–896

Knuth DE (1973) The art of computer programming, vol 1. Fundamental algorithms (2nd edn). Addison-Wesley, Reading, MA

Lauzon JP, Mark DM, Kikuchi L, Guevara JA (1985) Two-dimensional run-encoding for quadtree representation. Comput Vision Graph Image Proc 30(1):56–69

Matsuyama T, Hao LV, Nagao M (1984) A file organization for geographic information systems based on spatial proximity. Comput Vision Graph Image Proc 26(3):303–318

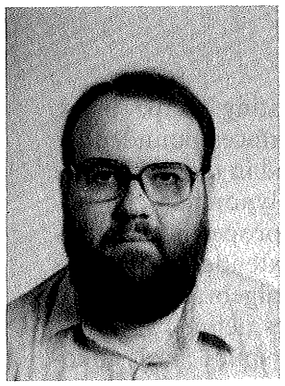
Meagher D (1982a) Geometric modeling using octree encoding. Comput Graph Image Proc 19(2):129–147

Meagher D (1982b) The octree encoding method for efficient solid modeling. Electrical and Systems Engineering Rep IPL-TR-032, Rensselaer Polytechnic Institute, Troy, NY (August 1982)

Nelson RC, Samet H (1987) A population analysis for hierarchical data structures. Proc SIGMOD Conf, San Francisco (May 1987), pp 270–277
Okawara F, Shimizu K, Nishitani Y (1988) Data compression of the region quadtree and algorithms for set operations. Dept Comput Sci Rep CS-88-6, Gunma Univ, Gunma, Japan (July 1988) (translated from Proc 36th All-Japan Conf on Information Processing, Information Processing Society of Japan, Tokyo, Japan (March 1988) pp 73–74)
Oliver MA, Wiseman NE (1983) Operations on quadtree-encoded images. Comput J 26(1):83–91
Orenstein JA (1982) Multidimensional tries used for associative searching. Inf Proc Lett 14(4):150–157
Robinson JT (1981) The k-d-B-tree: a search structure for large multidimensional dynamic indexes. Proc SIGMOD Conf, Ann Arbor, Michigan (April 1981), pp 10–18
Roussopoulos N, Leifker D (1985) Direct spatial search on pictorial databases using Packed R-trees. Proc SIGMOD Conf, Austin, Texas (May 1985), pp 17–31
Samet H (1984) The quadtree and related hierarchical data structures. ACM Comput Surv 16(2):187–260
Samet H (1985) Data structures for quadtree approximation and compression. Commun ACM 28(9):973–993
Samet H, (1990a) The design and analysis of spatial data structures. Addison-Wesley, Reading, MA
Samet H (1990b) Applications of spatial data structures: Computer graphics, image processing, and GIS. Addison-Wesley, Reading, MA
Samet H, Tamminen M (1985) Bintrees, CSG trees, and time. Comput Graph 19(3):121–130
Samet H, Tamminen M (1988) Efficient component labeling of images of arbitrary dimension represented by linear bintrees. ZEEE Trans Pattern Anal Mach Intell 10(4):579–586
Samet H Webber RE (1985) Storing a collection of polygons using quadtrees. ACM Trans Graph 4(3):182–222
Samet H, Rosenfeld A, Shaffer CA, Webber RE (1984) A geographic information system using quadtrees. Pattern Recognition 17(6):647–656
Samet H, Shaffer CA, Nelson RC, Huang YG, Fujimura K, Rosenfeld A (1987) Recent developments in linear quadtree-based geographic information systems. Image Vision Comput 5(3):187–197
Tamminen M (1984) Comment on quad- and octtrees. Commun ACM 27(3):248–249
Yau M, Srihari SN (1983) A hierarchical data structure for multidimensional digital images. Commun ACM 26(7):504–515

Received June 29, 1988; revised December 15, 1988

HANAN SAMET received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. Degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.
In 1975 he joined the Computer Science Department at the University of Maryland, College Park, where he is now a Professor. He is a member of the Computer Vision Laboratory of the Center for Automation Research and also has an appointment in the University of Maryland Institute for Advanced Computer Studies.
His research interests are data structures, computer graphics, geographic information systems, computer vision, robotics, programming languages, artificial intelligence, and database management systems. He is the author of the books *Design and Analysis of Spatial Data Structures*, and *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS* both published by Addison-Wesley, 1990.



ROBERT E. WEBBER received his Ph.D. in computer science from the University of Maryland at College Park. In 1983, he joined the Department of Computer Science at Rutgers University in New Brunswick, where he is now an Assistant Professor. His research interests are image synthesis, geographic information systems, analysis of algorithms, and discrete geometry. He is member of ACM, SIGGRAPH, IEEE, and NCGA.