Pergamon

Information Systems Vol. 20, No. 6, pp. 483–500, 1995
Copyright© 1995 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0306-4379/95 $9.50 + 0.00

0306-4379(95)00026-7

DYNAMIC INVERTED QUADTREE: A STRUCTURE FOR PICTORIAL DATABASES

MICHAEL VASSILAKOPOULOS[1] and YANNIS MANOLOPOULOS[2]

[1]Dept. of Electrical & Computer Eng., Aristotelian University of Thessaloniki, 540 06 Thessaloniki, Greece
[2]Dept. of Informatics, Aristotelian University of Thessaloniki, 540 06 Thessaloniki, Greece

(Received 30 July 1994; in final revised form 24 July 1995)

Abstract — We present a new structure, able to support and index a pictorial database. It is a variation of the region quadtree, termed Dynamic Inverted Quadtree, which is suitable for answering queries based on image content (exact and fuzzy image pattern searching). It exhibits certain advantages over the Fully Inverted Quadtree, an analogous structure that can be used for the same purposes. On the average it requires far less disk space; image pattern searching is performed efficiently; it is dynamic; reorganization is not obligatory, it can take place at any status of the structure and can be done on-line. Apart from the presentation of the new structure, we describe the pattern searching algorithms and we analyze the average space of the structure. Moreover, we compare analytically the space needs of Fully and Dynamic Inverted Quadtrees, as well as two different storage order methods that may be used in these structures. Analysis is based on a popular model of image randomness, which we express formally as a branching process.

1. INTRODUCTION

The role of Pictorial Databases is increasing in many modern applications. Medical research and practice, photo-journalism, art and fashion, industry, multimedia and many other activities of our age gain a lot from an information system supporting storage and retrieval of large numbers of digitized images [7].

The key issue in such an application is the kind of queries the related pictorial database can support. Some existing systems index images by alphanumeric (symbolic) descriptors. In particular, a caption is associated with each image. Queries based on (parts of) this caption can be performed. Other systems allow indexing by image content of specific kind [5]. More precisely, a set of features describing color, shape and texture (or even some more such general characteristics) are extracted from each image. These features are transformed under a distance preserving transform (e.g. FFT). A small number of the first coefficients of this transform are organized in a structure belonging in the family of R-trees and are, thus, used for indexing the image database. Queries on image content related to shape, texture and color can be applied. Due to the selective use of a limited number of coefficients, the output of such a query contains (in general) a superset of the images satisfying the query criteria. The queries that can be answered by such a system concern features for whole images. A similar system for 1-d data items that can satisfy queries about features of data sub-items has been presented in [6].

Nevertheless, there are applications that demand indexing and querying-by-content in an image database by directly handling the image bitmap representation. A popular structure for representing such bitmaps is the region quadtree (see [10] for a detailed presentation of this and other structures). Two region quadtree variations, suitable for storing sets of images in secondary memory, are the Multivalued Quadtree [1] and the Fully Inverted Quadtree [2]. The former structure permits the implementation of the geometric operators and topological predicates that are used in an SQL-based GIS [1]. The latter structure has introduced some truly novel ideas and is closely related to our subject, since it permits the implementation of exact and fuzzy image pattern searching within its image set [2]. More specifically, the FI-quadtree consits of a full quadtree, that is, a quadtree where each node has four children, except for the level-0 nodes. Each node holds a bit string of maximum length (the maximum number of images in the database). Each bit designates a separate image. The block corresponding to a particular node of the FI-quadtree is black for every image identified (by a 1) in the bit string of this node. This structure is kept entirely on secondary memory and accessed using a variation of hashing that is suitable for pattern searching. However, due to the large number of 0 bits in the bit strings, a lot of memory space is wasted. Furthermore, this structure is static in the sense that it can hold a predefined number of images. In case this number needs to be increased it may only be doubled and under the cost of total reorganization.

In this paper we present an improved inverted region quadtree variation, termed Dynamic Inverted (DI) Quadtree, where each node holds a list of identifiers of those images that have the corresponding block black, only. This structure supports the same kind of queries that its fully inverted cousin does. The FI-quadtree demands storage that is predefined and independent of the specific set of images, while the new structure requires storage that depends on the number of images, as well as on the specific data that these images induce. On the average (and not only) case the storage required by this structure is far less. In addition, it has been designed so that image pattern searching is done efficiently (thanks to an improved storage order). The new structure is dynamic in the sense that any number of images can be added. Reorganization is not obligatory, although it does improve performance. In addition, it can take place after any number of insertions and it may be performed up to any point and resumed later. Due to these characteristics, reorganization can be done on-line, during time periods where the system is not being updated or queried. The algorithms for fuzzy and exact pattern searching are similar for the two structures.

Moreover, we present a formal definition of a random region quadtree as a branching process. Setting specific probabilities for the different choices at each branching step, we succeed to express formally Samet and Shaffer's famous random model [9, 11]. Based on that model, we analyze the expected space occupied by a DI-quadtree and compare the storage requirements of the two structures. We also compare analytically the storage order method used in FI-quadtrees to the new storage order method of DI-quadtrees.

2. FULLY INVERTED QUADTREE

Consider class-n region quadtrees (that is, trees corresponding to images of $2^n \times 2^n$ pixels) and that we plan to store I_{MAX} ($= 2^i$ for a natural number i) images at maximum. The maximum number of nodes such a quadtree can have at level i equals 4^{n-i}. Summing up for all levels, we have $(4^{n+1} - 1)/3$ nodes for a full quadtree. The FI-quadtree for a set of images is formed by reserving contiguous disk space for $(4^{n+1} - 1)/3$ node-positions each one having I_{MAX} bits. The I_{MAX} bits at a specific node-position are used as flags, each one corresponding to a different image. The quadtree node represented by this position is black for all images whose flag is set. In fact, the FI-quadtree is a huge table of flags on disk. This structure is termed inverted since nodes store image identifiers (through flags) and not image data. A rather trivial FI-quadtree representing a database of four images appears in Figure 1. The size of an image is 2×2 pixels. At each node of the tree, we depict the corresponding set of raised flags. Although this is not shown in the figure, the order of the node-positions on disk follows a preorder traversal of the full quadtree.

Suppose we want to add the newly arrived j-th image. First, we encode it as a set of quadtree black-node prefixes which form a linear quadtree. If $j < I_{MAX}$ for each prefix we calculate a special hashing function which gives us an index to the appropriate node-position on disk. At this position we set the j-th flag. If, however, we have already stored I_{MAX} images, we have to copy the data of our FI-quadtree to a new FI-quadtree being able to store $2I_{MAX}$ images. After this total reorganization we may continue the insertion of the new image.

In [2], the process which is used for encoding the new image to a set of prefixes guaranties that the prefixes are produced in the same order as FI-quadtree node positions appear on disk. In other words, the prefixes for the black nodes of the new image are produced following the preorder traversal of the quadtree representing this image. According to the inventors of the FI-quadtree, this property results in few disk accesses. Although their remark is generally correct, the choice of preorder as "the same order" is not very good. In a quadtree, if a node is black then it is not possible for anyone of its descendants to be black too (e.g. nodes 1, 2 and 3 in Figure 2). Using preorder traversal as storage order, the only nodes which can be simultaneously black and appear consecutive on disk are sibling pixels (e.g. nodes 8 and 9, 9 and 10 or 10 and 11 in Figure 2) or

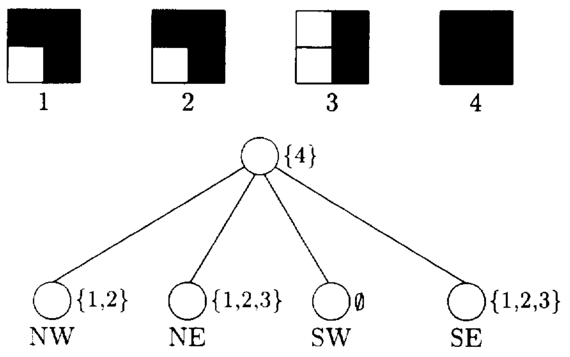

Fig. 1: An example demonstrating the logical aspects of an FI-quadtree.

a south-east pixel and a node at a higher level (e.g. nodes 16 and 17 in Figure 2). Insertion of two consecutive prefixes that correspond to two sibling level-i ($i > 0$) black nodes (e.g. nodes 2 and 7 in Figure 2), a very probable event, may result to accessing two remote disk pages, since on disk there is reserved space for all node positions of four class-(i–1) full subtrees between these two node positions.

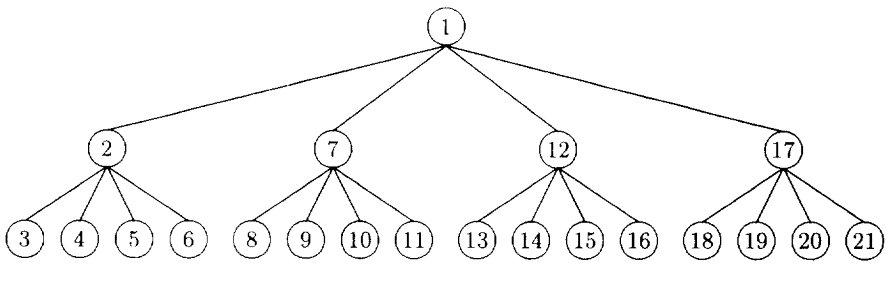

Fig. 2: Preorder traversal of a class-2 full quadtree.

Another drawback of the FI-quadtree is that it does not use the space efficiently: only a small fraction of the flags that correspond to an image will be set (at least for the great majority of images). Previous work [12] has shown that, on the average case, different quadtree levels vary significantly as far as the fraction of black nodes at a certain level is concerned.

Thus in summary, the FI-quadtree has the following drawbacks:

- poor storage utilization (since it is a static structure),
- storage method that limits the searching efficiency, and
- requires reorganization (after a certain number of insertions).

In the next section, we describe the DI-quadtree, which outperforms the FI-quadtree in all the above respects.

3. DYNAMIC INVERTED QUADTREE

The new structure we developed (termed Dynamic Inverted (DI) Quadtree) is a new quadtree variation. Consider again class-n region quadtrees and that we plan to store I_{MAX} images at maximum (with the difference that I_{MAX} may be very large now, e.g. 32,000 images).

3.1. Description of the Structure

The DI-quadtree has two parts: the *rear structure* and the *front structure*. The rear structure is a file of lists. The front structure is a full quadtree, where each node is a pointer to a rear structure list. Each list holds a number of distinct image identifiers. The quadtree node pointing to such a list is black for all images whose identifiers appear in this list.

The front structure is quite small and may even be held in the main memory of a modern computer. When a pointer of the front structure takes nil value the related list of the rear structure is empty (not existent). This value guards us from unnecessary accesses to the rear structure. A list of the rear structure is organized as a list of segments, where each segment may hold up to a number of image identifiers. Moreover, a segment contains a pointer to the next segment in the related list. This pointer is nil for the last segment of this list. In case this last segment is not full, the end of the list within this segment is denoted by repetition of the last image identifier. A rather trivial DI-quadtree demonstrating the same database of four images as in Figure 1 appears in Figure 3. The list segments hold two identifiers, at most. A DI-quadtree list is similar to a

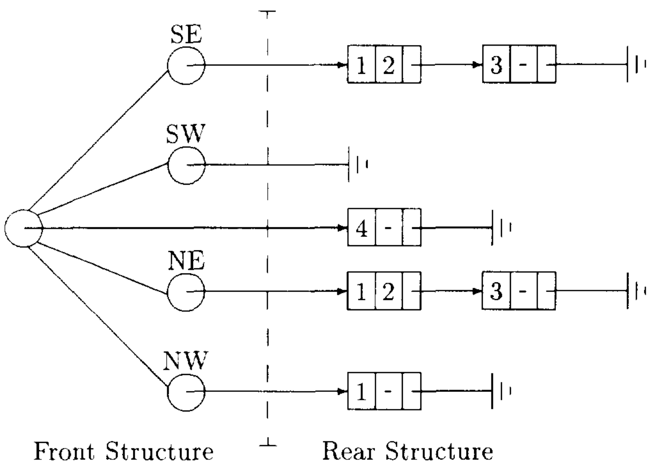

Fig. 3: An example demonstrating the logical aspects of a DI-quadtree.

"Fixed Size Posting Records List (FCHAIN)" which is discussed in [4]. In [4] the organization of an index over a key-attribute that has the same value in multiple records is examined: a B-tree is built for the distinct attribute values; each value points to a postings list which includes the locators of all the records that share this attribute value. In [4], four different list organizations are presented and analyzed when the distribution of attribute values is skewed.

The order of the pointers of the front structure follows the breadth-first traversal of the full quadtree. This is the order of the lists of the rear structure, after reorganization has been performed and no new images have been added to our database. Recall that the breadth-first traversal for a quadtree, first visits the root, then all nodes at level n–1 from left to right, then all nodes at level n–2 from left to right, ...and finally, all nodes at level 0 from left to right. Note that with this storage order, any two nodes whose lists appear consecutive on disk can be simultaneously black in a quadtree, with the only exception of the root and the next node in breadth-first traversal (the north-west child of the root). The chosen storage order helps the minimization of disk accesses during updates and queries (see Section 4.6).

3.2. Reorganization

Note that the organization described does not demand contiguous disk space for the rear structure, although high fragmentation would decrease performance. After many insertions of new images have taken place, the lists of the rear structure will not follow the breadth-first order.

The system will still be working, although the time taken by difficult queries will have increased. Moreover, the segments in many lists will not be consecutive on disk. Reorganization is exactly the process of putting lists and segments within lists to the desired order. This process may be performed up to a point, be suspended and resumed later (if insertions that jeopardize the order of the already reorganized part do not take place). This is why reorganization can be performed transparently, during time periods that the system is not queried or updated and be suspended at any time the user decides to interact with the database.

Note that I_{MAX} can also be changed (increased) a few times during the lifetime of the pictorial datatabase. Such a change increases, in general, the number of bits used for an image identifier. This means that the image identifiers already present in the database would be padded with zeros.

Note also (as the Analysis section justifies) that in the process of inserting more and more images in the database the capacity of the list segments might need to be changed, in order to reduce the percentage of non-data space in a segment.

Reorganization due to a change of I_{MAX} and/or of the segment capacity, if carefully programmed, again may be performed up to a point, be suspended and resumed later. Furthermore, such a change might require the increase of the number of bits used for a front structure pointer (see Section 4.1).

3.3. Insertion of New Images

The insertion of an image to the database consists of the following steps. First, the image is encoded to a set of prefixes for black nodes. Then, for each prefix, the related pointer of the front structure is accessed and the identifier of this image is inserted to the related list. Insertion of the identifier may result in augmentation of the list by one segment. In order to minimize the disk accesses, the prefixes to which this image is encoded must be in the same order (breadth-first) as the elements of the front and rear structure. This is accomplished if each prefix has two parts, the leftmost being the depth of the node and the rightmost the locational code of the upper left corner of the node (padded with zeros so as to be of fixed length $2n$). The desired order is produced by sorting the prefixes in ascending order. A pointer representing the j-th node at level i is found at position $(4^{n-i} - 1)/3 + j$ of the front structure. Note that a more intelligent batch insertion algorithm might also be devised.

3.4. Content Oriented Retrieval

Content oriented retrieval in a DI-quadtree pictorial database is a method for answering queries of the form: *"Given a rectangular subimage pattern having at most $2^n \times 2^n$ pixels, find all the images in the database that contain this pattern"*. The phrase "contains the pattern" used for an image in the database means that an equal size area of this image is completely identical to the pattern (exact search) or similar to the pattern (fuzzy search). Similarity between an area of an image and the pattern is measured by both the fraction of pattern prefixes that match and the fraction of pattern black pixels that match. We may allow the matching area to be at any possible position of the image grid, or to restrict it to a limited part of the grid. We will not discuss the latter option here, although both options are alike.

The searching algorithm for the DI-quadtree is analogous to that for the FI-quadtree. Let us give a description of pattern searching, adapted to the DI-quadtree, in brief:

- we translate this subimage to all the possible positions in the $2^n \times 2^n$ grid;
- for each position we encode the resulting image to a set of prefixes in breadth-first order; the set of matching images, M, is initially the set of all images present in the database;
- for each prefix we access the related list and intersect the set of images present in the list with M;
- after the last prefix of the certain position has been processed M contains the set of images that match the given subimage put at this position.

Fuzzy searching is an extension of this process where we take into account a matching factor, also. This factor is called *filtering ratio* of subimage pattern Z and of database image I:

$$d(Z, I) = \frac{1}{2}\left(\frac{N_{pref}}{Z_{pref}} + \frac{N_{pix}}{Z_{pix}}\right)$$

where Z_{pref} is the number of prefixes of the subimage translated at the position under examination, Z_{pix} is the number of black pixels these prefixes represent, N_{pref} is the number of these prefixes that match image I and N_{pix} the number of black pixels these matching prefixes represent. The filtering ratio expresses the kind of similarity described above: it contains (with equal weights) both the fraction of pattern prefixes that match and the fraction of pattern black pixels that match. The first fraction expresses similarity in terms of hierarchical decomposition to regular blocks, while the second fraction expresses similarity in terms of area of the black regions. The steps of fuzzy pattern searching would be now:

- we translate this subimage to all the possible positions in the $2^n \times 2^n$ grid;
- for each position we encode the resulting image to a set of prefixes in breadth-first order; we correspond with each image of the database two variables N_{pref} and N_{pix}, initialized to 0.
- for each prefix we access the related list and modify N_{pref} and N_{pix} for all images present in this list;
- after the last prefix of the certain position has been processed we can calculate $d(Z, I)$ and estimate the similarity to the image pattern for all images present in the database.

If our images are similar (that is, they have large common black regions) the accesses take place to a limited number of lists being close to each other. This means that in the case of similar images pattern searching is considerably speeded up. In addition, we have substantial space reduction since there are not many segments in lists that contain a few entries.

Note that the fuzzy searching desribed above examines only the matching between black nodes of the pattern and black nodes of images in the database that belong at the same quadtree level. A more sophisticated fuzzy searching should also consider the case where some images of the database contain nodes that cover a black node of the pattern (since they correspond to ancestors of this node) or that are covered by a black node of the pattern (since they correspond to descendants of this node). For each prefix of the patern, examination of prefixes at the previous and at the next level would give sufficiently accurate results.

Note also that, since the displacement of an image is a very costly operation when involving a quadtree generation, the searching and translation operations can be done simultaneously [2]. Consider a number of rear structure lists that have been transferred in main memory. We may perform searchings for a set of image positions using these lists only. When a searching needs to access a list residing on disk it is temporarily interrupted and resumed later, when the requested lists will have been transferred in main memory. This strategy is expected to work very well (with few disk accesses) for a database of similar images due to the high locality of lists.

4. ANALYSIS

The definition of the basic parameters that appear in the rest of this section is given in Table 1.

4.1. Front Structure Storage Needs

In order to evaluate the storage requirements of the front structure, we have to estimate the pointer size, which is a function of the number of segments in the rear structure. Suppose that we have inserted all I_{MAX} images. The reader could easily see that the maximum number of black nodes a quadtree may have is the maximum number of black level-0 nodes. Since, only three out of four sibling leaf nodes can be simultaneously black, we conclude that the maximum

I	Number of images in the database at a specific instant
I_{MAX}	Maximum number of images in the database
P	Average number of disk pages needed by the rear structure
b	Bits needed by a front structure pointer or an inter-segment pointer
P_{cap}	Capacity of a disk page in segments
S_{cap}	Capacity of a segment in identifiers
S	Bound of the maximum number of segments the database may have
U	Storage utilization factor of FI-quadtrees
$p(l)$	Probability that a list has exactly l elements
$P(L)$	Probability that a list has $\leq L$ elements
n	Class of quadtrees (an image contains $2^n \times 2^n$ pixels)
Q_n	Set of all class-n quadtrees
S_i	Set of all class-i sub-quadtrees
x	Probability of each of the next two choices: the root to be black or white
y_i	Branching probability at level-i of a quadtree

Table 1: The fundamental parameters appearing in the paper

number of nodes of a class-n quadtree equals $3/4 \times 4^n = 3 \times 4^{n-1}$. If all I_{MAX} images have the maximum number of black nodes then we have $3I_{MAX} \times 4^{n-1}$ image identifiers. The worst way these identifiers may be distributed to segments is the case that any non-full segment contains only one identifier. Next we are going to estimate the number of segments for this (worst) case.

Suppose a segment can hold S_{cap} identifiers. For every pointer of the front structure (remember there are $(4^{n+1} - 1)/3$ pointers) we reserve a segment by inserting one identifier. We call such a segment "just-reserved" (it has $S_{cap} - 1$ free entries). An example of this situation, where we have reserved one segment for every pointer of the front structure, appears in Figure 4. Each segment contains one identifier and has $S_{cap} - 1$ empty entries. Note that this is possible for every pointer (we have enough identifiers, since, for $I_{MAX} > 1$, we have $3I_{MAX} \times 4^{n-1} > (4^{n+1} - 1)/3$). We

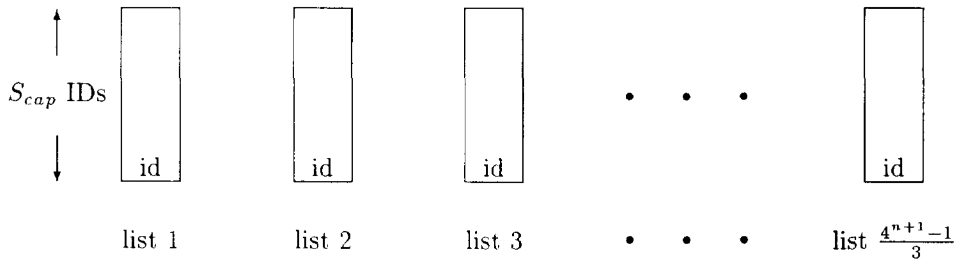

Fig. 4: A step of the segment reservation policy.

are then left with $3I_{MAX} \times 4^{n-1} - (4^{n+1} - 1)/3$ "unused" identifiers. By picking S_{cap} "unused" identifiers we can fill the $S_{cap} - 1$ free entries of one of the just-reserved segments and create a new just-reserved segment. We continue the same reservation policy until we are left with less than S_{cap} "unused" identifiers for which there is enough room in any one of the just-reserved segments. We conclude that the number of segments our rear structure might hold is bounded by

$$S = \frac{4^{n+1} - 1}{3} + \left\lfloor \frac{3I_{MAX} \times 4^{n-1} - (4^{n+1} - 1)/3}{S_{cap}} \right\rfloor .$$

This means that

$$b = \lceil \log(S+1) \rceil$$

bits are enough for a pointer (we add 1 for the nil pointer value). Thus, the front structure occupies exactly

$$b \times (4^{n+1} - 1)/3$$

bits of space in main (or secondary) memory. We can reach a simpler approximate formula for b since $(4^{n+1} - 1)/3 \approx 4^{n+1}/3$. Then,

$$\begin{aligned} S &\approx \frac{4^{n+1}}{3} + 4^{n-1}\frac{9I_{MAX} - 4^2 + 1}{3S_{cap}} \\ &\approx \frac{4^{n+1}}{3} + 4^{n-1}\frac{9I_{MAX}}{3S_{cap}} \\ &= 4^{n-1}\frac{9I_{MAX} + 4^2 S_{cap}}{3S_{cap}} \end{aligned}$$

Since $\lceil \log(S+1) \rceil \approx \lceil \log S \rceil$ we have that

$$b \approx \lceil 2(n-1) + log(9I_{MAX} + 4^2 S_{cap}) - log(3S_{cap}) \rceil \tag{1}$$

For example, setting $n = 10$, $I_{MAX} = 1024$, $S_{cap} = 15$ we find that $b = 26$ bits and that the front structure needs 4.33 Mbytes of space. Setting $n = 10$, $I_{MAX} = 16384$, $S_{cap} = 15$ we find that $b = 30$ bits and that the front structure needs 5 Mbytes of space. In both cases, a modern computer could keep this structure in main memory.

4.2. *Probabilistic Model*

Shaffer and Samet have given a descriptive definition of a model of random quadtrees [9, 11]. According to this model *"each leaf node is assumed to be equally likely to appear at any position and level in the tree"*. This model is generally considered a very realistic one for images usually found in practice and it has been used for the analysis of neighbor finding algorithms producing results close to statistics of real tests. However, there has never been presented a formal equivalent of the above definition. We give such a definition in the rest of this section. In order to do that, we first examine the set-theoretic representation of quadtrees.

Note that we can view a region quadtree as a finite ordered tuple that consists of gray, black and white nodes and corresponds to the preorder traversal of the tree. With this in mind, we can devise a symbolic way to represent the set of all class-n quadtrees (a finite set having as elements tuples) as has been done for other trees and combinatorial structures [13].

We shall use the notation $< T1, T2, T3, T4 >$ to denote all the distinct sub-quadtree configurations that can be formed by a gray root and the sub-quadtrees $T1$, $T2$, $T3$ and $T4$ as its children. The number of these configurations equals the number of the distinct permutations of $T1$, $T2$, $T3$ and $T4$. Since each such configuration denotes a set of sub-quadtrees, the notation $< T1, T2, T3, T4 >$ represents a union of sets of sub-quadtrees. An example showing the four different sub-quadtree configurations that are represented by the notation < ■□□□ > appears in Figure 5. If the + symbol represents set union and S_i the set of class-i subtrees ($n \geq i \geq 1$), then

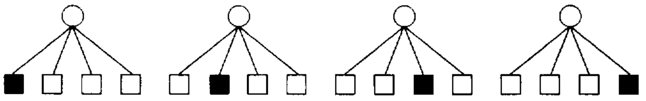

Fig. 5: The four sub-quadtree configurations represented by the notation < ■□□□ >.

the set of all class-n quadtrees, Q_n, is:

$$\begin{aligned}
Q_n &= \square + \blacksquare + S_n \\
S_i &= <\blacksquare\square\square\square> + <\blacksquare\blacksquare\square\square> + <\blacksquare\blacksquare\blacksquare\square> + \\
&\quad <S_{i-1}\square\square\square> + <S_{i-1}\blacksquare\square\square> + <S_{i-1}\blacksquare\blacksquare\square> + <S_{i-1}\blacksquare\blacksquare\blacksquare> + \\
&\quad <S_{i-1}S_{i-1}\square\square> + <S_{i-1}S_{i-1}\blacksquare\square> + <S_{i-1}S_{i-1}\blacksquare\blacksquare> + \\
&\quad <S_{i-1}S_{i-1}S_{i-1}\square> + <S_{i-1}S_{i-1}S_{i-1}\blacksquare> + \\
&\quad <S_{i-1}S_{i-1}S_{i-1}S_{i-1}> \qquad \forall\, i > 1 \\
S_1 &= <\blacksquare\square\square\square> + <\blacksquare\blacksquare\square\square> + <\blacksquare\blacksquare\blacksquare\square>
\end{aligned}$$

In the tables that follow we depict the number of different sub-quadtree configurations each of the <> notations above represents. These are all the possible <> notations for configurations of class-i sub-quadtrees in the sense that any other such notation is equivalent to one of them.

$<\blacksquare\square\square\square>$	4
$<\blacksquare\blacksquare\square\square>$	6
$<\blacksquare\blacksquare\blacksquare\square>$	4

$<S_{i-1}\square\square\square>$	4
$<S_{i-1}\blacksquare\square\square>$	12
$<S_{i-1}\blacksquare\blacksquare\square>$	12
$<S_{i-1}\blacksquare\blacksquare\blacksquare>$	4
$<S_{i-1}S_{i-1}S_{i-1}S_{i-1}>$	1

$<S_{i-1}S_{i-1}\square\square>$	6
$<S_{i-1}S_{i-1}\blacksquare\square>$	12
$<S_{i-1}S_{i-1}\blacksquare\blacksquare>$	6
$<S_{i-1}S_{i-1}S_{i-1}\square>$	4
$<S_{i-1}S_{i-1}S_{i-1}\blacksquare>$	4

Table 2: All possible <> notations and the number of different sub-quadtree sets each one represents.

The above symbolic equations describe a branching process by which we can construct any legal class-n region quadtree. More specifically,

- At the beginning, we perform the *initial branching*: we choose between the tree root being a black node, a white node or a gray node (root of a class-n sub-quadtree).

- At any level i, $n \geq i > 1$, for any gray node at this level, we perform a *level-i branching*: we choose between 79 different subtree configurations (those of all the three tables above) and set this node to be the root of the chosen configuration.

- At level 1, for any gray node at this level, we perform a *level-1 branching*: we choose between 14 different subtree configurations (those of the leftmost table above) and set this node to be the root of the chosen configuration.

Note that except for the initial branching, where we choose between different colors, at all other branchings we choose between different subtree configurations. This approach is different to the branching process described in [8], where at each node (starting at the root), we always choose between white, black and gray colors; if we color a node gray, we continue the process recursively for each one of its children choosing always between all these three colors. Although the approach of [8] is simpler, it cannot model the usual quadtrees (called condensed quadtrees in [8]), since it allows tree configurations where all four children of a gray node are leaves of the same color.

We can create a probabilistic model for this branching process by assigning probabilities to the different choices for every branching. This process must be legal under the fundamental probability axioms. Thus, for every specific branching the probabilities of all the different choices must sum to 1.

We would like to find a simple assignment of probabilities to the different choices of our branching model that would lead to a formal definition of Samet and Shaffer's model. Remember that at the initial branching we have three choices: the root to be black, white or gray. We call x the probability each of the first two choices. Then, the probability of the third choice will be $1 - 2x$. Remember also that at all levels above level 1 we have 79 choices. We would like all of them except for $< S_{i-1}S_{i-1}S_{i-1}S_{i-1} >$ to be equiprobable (for the sake of simplicity). We call y_i the probability of each of these 78 choices. Since the sum of the probabilities of all the 79 choices must be 1, the probability of $< S_{i-1}S_{i-1}S_{i-1}S_{i-1} >$ will be $1 - 78y_i$. Accordingly, at level 1 we would like all 14 choices to be equiprobable. In Table 3 we depict this simple parametric assignment of probabilities for the different choices at each branching. We notice again that each <> notation corresponds to many isomorphic subtree configurations and its probability is the probability of each of these different subtree choices. Let us call this branching process along with this parametric assignment of branching probabilities the *"new random quadtree model"*. If we can find values, positive and smaller than 1, for x and y_i such that the probability of a black or white node existing anywhere in the tree is constant, then we shall have proved that the new random quadtree model is equivalent to Samet and Shaffer's descriptive model.

	Initial Branching	level-i ($i > 1$) Branching	level-1 Branching
□	x		
■	x		
S_n	$1 - 2x$		
< ■□□□ >		y_i	1/14
< ■■□□ >		y_i	1/14
< ■■■□ >		y_i	1/14
$< S_{i-1}$□□□ $>$		y_i	
$< S_{i-1}$■□□ $>$		y_i	
$< S_{i-1}$■■□ $>$		y_i	
$< S_{i-1}$■■■ $>$		y_i	
$< S_{i-1}S_{i-1}$□□ $>$		y_i	
$< S_{i-1}S_{i-1}$■□ $>$		y_i	
$< S_{i-1}S_{i-1}$■■ $>$		y_i	
$< S_{i-1}S_{i-1}S_{i-1}$□ $>$		y_i	
$< S_{i-1}S_{i-1}S_{i-1}$■ $>$		y_i	
$< S_{i-1}S_{i-1}S_{i-1}S_{i-1} >$		$1 - 78y_i$	

Table 3: A parametric assignment of probabilities for the different choices at each branching.

Note that

- The set of all class-n quadtrees is a probability space.
- Each quadtree is a distinct outcome in this space.
- The probability of a specific quadtree is the product of the probabilities of all the choices we made at each level in order to built it.
- The probability of any subset of quadtrees equals the sum of their probabilities.

We can now prove the following theorem:

Theorem 1 *If $x = 1/(2n + 2)$ and $y_i = 1/52i$, the new random quadtree model is equivalent to Samet and Shaffer's random tree model.*

Proof. The case of black nodes is examined. Due to symmetry the same arguments hold for white nodes, too. Consider all trees that include a black node in a certain level and position. The probability of this tree set must be constant, equal to a value c $(1 \geq c > 0)$ for any level and position of this node.

Figure 6 shows an abstract example of the branchings we perform at each level when we construct quadtrees that have a certain node black. The plain dotted rectangles stand either for a

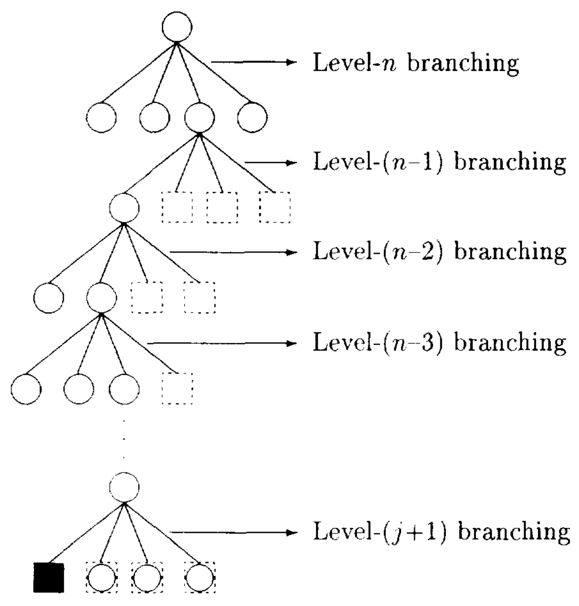

Fig. 6: An abstract example of the branchings we perform when constructing quadtrees with a certain black node.

black or a white node; the dotted rectangles that include a circle stand either for a black, a white, or a gray node. Note that at the last branching a configuration with four black nodes cannot exist.

Each of these trees contains the path leading to the node in question. At each level above this node there is a set of choices we may make in order to construct this path. The probability of such a set of choices equals the sum of their probabilities (let us call it "level-i choice-probability"). Since we must consider all possible ways of constructing quadtrees that have this node colored black, the probability of our tree set equals the product of the level-i choice-probabilities, for all levels above the specific node. We distinguish three cases for the level of this node: first, to be at level n (the root); second, to be at any level, j, below n and above 0 (not the root or a pixel); and, third, to be at level 0 (a pixel). The probability of our tree set (according to each case) is:

1: the probability of the root being black, x; we must have

$$x = c \tag{2}$$

2: the product of the following probabilities:
a) the probability of the root being gray ($1 - 2x$, initial branching).
b) for each level above the parent level of our black node, the probability of all the choices in which a certain node (the one being on the path) is gray. For every level there is

- 1 such choice where all four siblings are gray (prob. $= 1 - 78y_i$, e.g. level-n branching in Figure 6: the 3 nodes that are not on the path are all gray),
- 8 such choices where only one of the four siblings (the one on the path) is gray (prob. $= 8 \times y_i$, e.g. level-$(n-1)$ branching in Figure 6: each of the 3 leaves that are not on the path can be black or white),
- 12 such choices where two of the four siblings are gray (prob. $= 12 \times y_i$, e.g. level-$(n-2)$ branching in Figure 6: the gray node not on the path can take 3 positions, while each of the 2 leaves that are not on the path can be black or white) and
- 6 such choices where three of the four siblings are gray (prob. $= 6 \times y_i$, e.g. level-$(n-3)$ branching in Figure 6: the leaf that is not on the path can take 3 positions, while at each position it can be black or white).

For each level i, $n \geq i > j + 1$, the total probability of these choices is $1 - 52y_i$.

c) for the parent level of the node in question, the probability of all the choices in which this node is black. The choices are 26 and their total probability is $26 \times y_{j+1}$ (e.g. level-(j+1) branching in Figure 6: each of the 3 nodes that are not on the path can be gray, white, or black, excluding the illegal configuration where all of them are black).
Our equation is now

$$(1-2x)(1-52y_n)(1-52y_{n-1})\cdots(1-52y_{j+2})26y_{j+1}=c \quad n-1 \geq j \geq 1 \tag{3}$$

In fact this is a system of $n-1$ equations.

3: the probabilities we must multiply are the almost same as in case 2; the only difference is that the choices at level 1 (the parent level of the black node) in which this node is black are 7 and their total probability is $7 \times 1/14 = 1/2$. Thus, since $j = 0$ now, our equation is

$$(1-2x)(1-52y_n)(1-52y_{n-1})\cdots(1-52y_2)1/2=c \tag{4}$$

Equations 2,3 and 4 form a system of equations that expresses the conditions under which the *new random quadtree model* is equivalent to Samet and Shaffer's model. This system may be solved as follows: we substitute the value of c given by Equation 2 to Equations 3 and 4. Solving Equation 3 for $j = n-1$, we find y_n; solving it for $j = n-2$ and using the value of y_n, we find $y_{n-1}, \ldots$; solving it for $j = 1$ and using the values of $y_n \ldots y_3$, we find y_2. In general we conclude that

$$y_i = \frac{1}{26}\,\frac{x}{1-2(n-i+1)x} \tag{5}$$

For all $i > 1$, substituting the value of y_i from the above equation to 4 we find that $x = 1/(2n+2)$. Substituting this value to Equation 5 we find that $y_i = 1/52i$. □

As the proof above justifies, the new random quadtree model includes the Samet and Shaffer's random tree model as a specific case (for the simple parametric assignment of probabilities of Table 3 and the specific values of x and y_i). However, the introduction of this model has a wider importance. The new model might prove appropriate for other kinds of interesting images, depending on the assignment of probabilities. Moreover, it could be further extended if the all the level-i sub-quadtree configurations are not treated equivalently. Then it could express images that do not follow this kind of symmetry (for example, images where the probability of a pixel being black depends on its distance from a specific pixel, called the source of black).

4.3. Rear Structure Average Space Requirements

Suppose that at a specific moment I images have been inserted in our database. What is the expected number of pages, P, needed by the rear structure?

Let $p(l)$ be the probability that a list has exactly l elements. A list will have l elements, if l out of the I images have the corresponding node black and the rest $I-l$ do not (Bernoulli trials):

$$p(l) = \binom{I}{l}\left(\frac{1}{2n+2}\right)^{l}\left(1-\frac{1}{2n+2}\right)^{I-l}$$

Let $P(L)$ be the probability that a list has $\leq L$ elements. It is clear that

$$P(L) = \sum_{l=0}^{L} p(l) = \left(\frac{2n+1}{2n+2}\right)^{I} \sum_{l=0}^{L} \binom{I}{l}\left(\frac{1}{2n+1}\right)^{l}$$

If $R(i)$ gives the probability that a list has i segments then:

$$R(i) = P(iS_{cap}) - P\left((i-1)S_{cap}\right)$$

Since $\binom{I}{l} = 0$ when $l > I$, the expected number of segments in a list is

$$\sum_{i=1}^{\lceil I/S_{cap}\rceil} iR(i) = \left(\frac{2n+1}{2n+2}\right)^{I} \sum_{i=1}^{\lceil I/S_{cap}\rceil} i \sum_{j=1}^{S_{cap}} \binom{I}{(i-1)S_{cap}+j} \left(\frac{1}{2n+1}\right)^{(i-1)S_{cap}+j}$$

or, eliminating terms equal to 0 (treating the last segment differently), the expected number of segments in a list is

$$\left(\frac{2n+1}{2n+2}\right)^{I} \left[\sum_{i=1}^{\lceil I/S_{cap}\rceil-1} i \sum_{j=1}^{S_{cap}} \binom{I}{(i-1)S_{cap}+j} \left(\frac{1}{2n+1}\right)^{(i-1)S_{cap}+j} + \right.$$
$$\left. \lceil I/S_{cap}\rceil \sum_{j=1}^{I-(\lceil I/S_{cap}\rceil-1)S_{cap}} \binom{I}{(\lceil I/S_{cap}\rceil-1)S_{cap}+j} \left(\frac{1}{2n+1}\right)^{(\lceil I/S_{cap}\rceil-1)S_{cap}+j} \right]$$

If a disk page can hold P_{cap} segments, then for the total expected number of pages needed by the rear structure we have

$$P = \left\lceil \frac{(4^{n+1}-1)\sum_{i=1}^{\lceil I/S_{cap}\rceil} iR(i)}{3P_{cap}} \right\rceil$$

Note that a segment cannot be split in two pages. Thus, if the page size is not analogous to the segment size, each page will have some empty space. It is obvious that the storage efficiency of the rear structure depends on the number of images inserted in the database and on the capacity of each list segment. This is also true for the storage efficiency of the whole tree, since the front structure is much smaller than the rear structure. As has been pointed out in Section 3.2, the segment capacity may be changed at any instant in the lifetime of the database. Thus, it is interesting to investigate what is a good choice for S_{cap} given the values of I and I_{MAX}. A measure for a certain S_{cap} value is the average number of non-data bits in a list. These non-data bits belong to three categories: the bits used for the pointer field in each segment, the empty entries of the last segment in a list and the possible empty space in each page. It is not difficult to develop formulae that calculate this non-data space in a list (for example, the average space used for pointers is the expected number of segments in a list multiplied by b).

We created Figure 7 for I_{MAX} equal to 4096 images and for various values of I. The page size equals 4096 bits (512 bytes). Although this is not demonstrated here, note that if a good choice for S_{cap} produces segments which are larger than the disk page size, we may enlarge the (logical) page size so as to consist of a small number of physical pages. The symbols on each y-axis (corresponding to $S_{cap} = 0$) demonstrate the number of data bits in a list for the respective number of images in the database. This diagram is just a sample among many diagrams we created for different I_{MAX} values. In these diagrams, for all practical cases (I_{MAX} ranging from more than 1024 and up to several tens of thousands of images), the minimum number of non-data bits is always less than 1/3 of the number of data bits in the respective list.

4.4. Storage Efficiency of the FI-quadtree

As we have already shown in Section 2, when the number of images already inserted in the database is at least $1 + I_{MAX}/2$ and at most I_{MAX}, the space requirements of an FI-quadtree equal $I_{MAX} \times (4^{n+1} - 1)/3$ bits of disk space. This space is independent of the data of these images.

Since the images in the database are considered independent to each other, the average number of raised flags in a disk position equals $I/(2n+2)$ and the average number of raised flags in the whole structure will be $\frac{I}{2n+2} \times \frac{4^{n+1}-1}{3}$. We call U the storage utilization factor, that is, the fraction

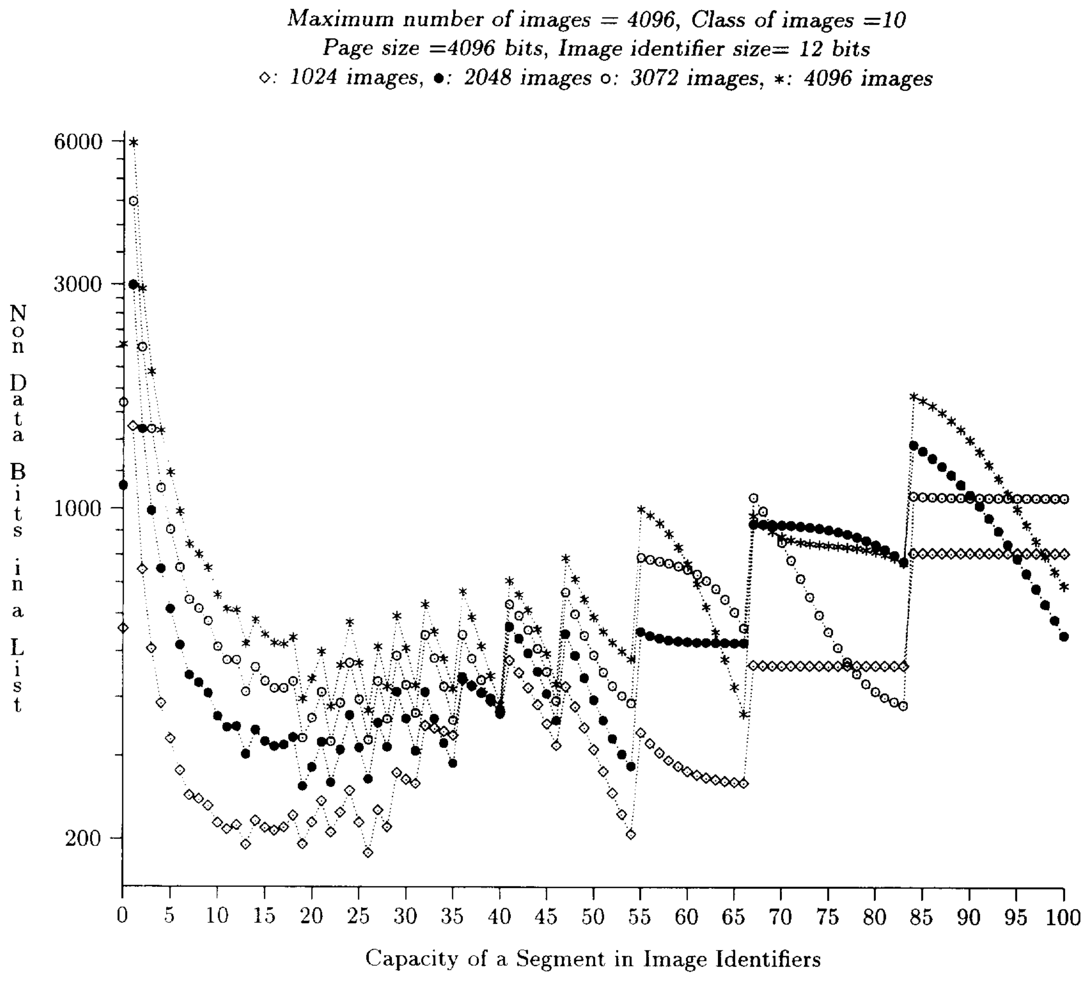

Fig. 7: Average number of non-data bits in a list as a function of the segment capacity.

of used space (space of raised flags) over the total space occupied by an FI-quadtree. We have that

$$U = \frac{I}{I_{MAX}(2n+2)}$$

This factor gets minimized when $I = 1 + I_{MAX}/2$ and equals

$$U_{min} = \frac{I_{MAX} + 2}{2I_{MAX}(2n+2)} \simeq \frac{1}{2(2n+2)}$$

It gets maximized when $I = I_{MAX}$ and equals

$$U_{max} = \frac{I_{MAX}}{I_{MAX}(2n+2)} = \frac{1}{2n+2}$$

For example, setting $n = 10$ we have $U_{min} = 2.27\%$ and $U_{max} = 4.55\%$.

4.5. Comparison of the Two Structures

The number of nodes of a full quadtree is equal to $(4^{n+1} - 1)/3 \simeq 4^{n+1}/3$. An FI-quadtree needs I_{MAX} bits for each node. Thus, its total space requirements are approximately $\frac{4^{n+1}}{3} \times I_{MAX}$ bits.

In a DI-quadtree, the rear structure has by far the biggest disk space requirement in comparison with the front structure. In order to represent the identifier of an image in the rear structure, we need $\lceil \log(I_{MAX}) \rceil \simeq \log(I_{MAX})$ bits. Since images are independent to each other, on the average case a list will have $I \times \frac{1}{2n+2}$ entries, or $I_{MAX} \times \frac{1}{2n+2}$ when all the I_{MAX} images have been inserted. We assume that the non-data space in a list (pointers and unused space in segments) is not more than 1/3 of the data space in this list. Since $1 + 1/3 = 4/3$ we can say that the total storage requirements for a list are in the order of $\frac{4}{3} \times I \times \frac{1}{2n+2} \times \log(I_{MAX})$ bits. Comparing this formula for the approximate size of a list with the formula for the approximate size of a front structure pointer (Equation 1) it is obvious that the list size is much larger than any of the three terms of Equation 1. We may then say that the space requirements of the DI-quadtree are in the order of $\frac{4}{3} \times \frac{4^{n+1}}{3} \times I \times \frac{1}{2n+2} \times \log(I_{MAX})$ bits.

We would like to investigate the conditions under which the DI structure needs far less memory than the FI structure. For any number of images between $I_{MAX}/2$ and I_{MAX} the FI-quadtree storage requirements are the same, since it is a static structure. However the DI-quadtree is a dynamic structure, and, therefore, its storage requirements (may) change even after the insertion of a single image. In order to be fair, we compare their storage requirements for the case of $I = \frac{3}{4} \times I_{MAX}$. We seek for the conditions under which

$$\frac{4}{3} \times \frac{4^{n+1}}{3} \times \frac{3}{4} \times I_{MAX} \times \frac{1}{2n+2} \times \log(I_{MAX}) << \frac{4^{n+1}}{3} \times I_{MAX}$$

The above inequality implies that

$$I_{MAX} << 2^{2n+2}$$

For example, if our images are 1024×1024 bits, then the DI-quadtree will be more space efficient than the FI-quadtree if the maximum number of images is far less than $2^{22} = 4,194,304$. In other words, the DI-quadtree is far more space efficient than the FI-quadtree for all practical cases.

This analytical comparison is supplemented by Table 4 where we show the absolute space requirements of both trees and the fraction of the DI-quadtree size over the FI-quadtree size for I_{MAX} equal to 512, 1024, 2048, 4096, 8192, 16384 and 32768 images. The number of images inserted in the database is always $I = \frac{3}{4} \times I_{MAX}$. Table 4 was produced by using the exact formulae giving the average sizes of DI and FI quadtrees. It is interesting to note that the demonstrated fraction is around 50% in all cases.

I_{MAX}	DI space (Mb)	FI space (Mb)	$\frac{\text{DI space}}{\text{FI space}}\%$
512	45.853	85.333	53.73
1024	85.484	170.667	50.09
2048	166.972	341.333	48.92
4096	337.507	682.667	49.44
8192	690.258	1365.333	50.56
16384	1425.966	2730.666	52.22
32768	2972.346	5461.332	54.43

Table 4: Comparison of DI and FI quadtree storage requirements.

4.6. Breadth-First versus Preorder

Given an FI-quadtree or a DI-quadtree, the choice of breadth-first or of preorder storage order affects pattern matching speed. The form of the black-node prefixes to which an image is encoded differs according to the storage order used. However, the number of these prefixes is the same for both orders.

Consider a pair of prefixes which appear consecutive in the encoding of an image pattern. Let us assume that, if the FI-quadtree or DI-quadtree lists which correspond to these prefixes, are consecutive on disk, then there is 1 unit of cost for accessing the two lists. On the contrary, when the two lists are not consecutive on disk, there are 2 units of cost for accessing these lists. We refer to this cost scheme for the whole image pattern as the "searching time cost". Note this cost scheme is virtual and should not be confused with the actual cost of pattern searching (the actual number of disk accesses). However, intuition says that a low virtual cost is quite strong evidence for a low actual cost, too: accessing the second of two consecutive lists requires one extra sequential disk access at most, while accessing the second of two non consecutive lists most probably requires one extra random disk access.

Note that, two prefixes may appear successively in an image encoding, only if the nodes to which they correspond are not ascenstor and descendant (only if they do not belong in the same path). We shall call such a pair of nodes (of prefixes) a legal pair of nodes (prefixes). Consider a large sequence of image patterns for which we search our database. Eventually, all possible legal pairs of prefixes appear in this sequence. More specifically, due to the law of large numbers, the frequency of appearence of each legal pair equals its probability of existence. The total searching time cost of the sequence will be two times the total number of legal pairs of prefixes in the sequence, minus the number of those legal pairs of prefixes whose lists appear consecutive on disk. We can now prove the following theorem.

Theorem 2 *In an FI-quadtree or DI-quadtree image database, the choice of Breadth-first instead of Preorder as storage order results in smaller average searching time cost.*

Proof. In this proof, let the term "node" be equivalent to the term "list corresponding to node" (for the sake of readability). As has been noted in Section 2 and Figure 2, when using preorder traversal as storage order the legal pairs of nodes which appear consecutive on disk are either two sibling pixels ($3 \times 4^{n-1}$ pairs) or a south-east pixel and a higher level node ($4^{n-1} - 1$ pairs). As far as pixel pairs are concerned, the same legal pairs of consecutive sibling pixels appear when using breadth-first traversal as storage order. If we prove that, for each of the rest $4^{n-1} - 1$ consecutive legal pairs of preorder traversal, there is a non-pixel consecutive legal pair of breadth first traversal that has higher probability of existence, then we will have proved the above theorem.

Let us start with preorder traversal. We seek for the probability of all the trees that have a black pixel and a black level-j node ($n > j > 0$) at certain positions. The branching process for creating such a pair is similar to the branching process of case 3 in the proof of Theorem 1. The only difference appears at level-(j+1) branching: each of the possible subtree choices should have, as children of the subtree root, a gray node (ancestor of the black pixel) and a black node (the level-j node under consideration) at specific positions. The other two children of the subtree root may be white, black or gray nodes. It is easy to see that these choices are 9 (2 nodes each one taking any of 3 colors count for 3^2 choices). Since each of them has a probability of $1/52(j+1)$ their total probability is $9/52(j+1)$. The branchings continue at levels lower than j+1 as usually. Multiplying the probabilities of the sets of choices we may make at all levels we conclude that the requested probability is

$$(1-2x)(1-52y_n)\cdots(1-52y_{j+2})\,\frac{9}{52(j+1)}\,(1-52y_j)\cdots(1-52y_2)1/2 =$$
$$\frac{n}{n+1}\,\frac{n-1}{n}\cdots\frac{j+1}{j+2}\,\frac{9}{52(j+1)}\,\frac{j-1}{j}\cdots\frac{1}{2}\,\frac{1}{2} = \frac{1}{2n+2}\,\frac{9}{52j} \qquad n > j > 0$$

It is maximized when $j = 1$ and equals $\frac{1}{2n+2} \times \frac{9}{52}$. The leftmost part of the above equalities is similar to the left part of Equation 4, except for the level-(j+1) probability.

We now turn to breath-first traversal. We seek for the probability of all the trees that have a certain pair of sibling black nodes at level-j ($n > j > 0$). The branching process for creating such a pair is similar to the branching process of case 2 in the proof of Theorem 1. The only difference appears at the last (level-(j+1)) branching: each of the possible subtree choices should have, as children of the subtree root, two black nodes at specific positions. The other two children of the subtree root may be colored white, gray, or one of them black. The choices now are 8 (2 nodes each one taking any of 3 colors, except for the case that they are both black count for $3^2 - 1$ choices);

their total probability equals $8 \times \frac{1}{52(j+1)}$ and the requested probability turns out to be

$$(1-2x)(1-52y_n)\cdots(1-52y_{j+2})\frac{8}{52(j+1)} =$$

$$\frac{n}{n+1}\,\frac{n-1}{n}\cdots\frac{j+1}{j+2}\,\frac{8}{52(j+1)} = \frac{1}{2n+2}\,\frac{8}{26} \qquad n > j > 0$$

The leftmost part of the above equalities is similar to the left part of Equation 3, except for the level-(j+1) probability.

Obviously, $\frac{1}{2n+2} \times \frac{8}{26} > \frac{1}{2n+2} \times \frac{9}{52}$. How many are these pairs in breadth-first traversal? At level-(n-1) there are 3×4^0, at level-(n-2) there are 3×4^1 ... at level-1 there are $3 \times 4^{n-2}$ pairs. The sum of all these is $4^{n-1} - 1$, exactly as many as the preorder pairs that consist of a south-east pixel and a higher level node. □

It is worth noting that, at each level below level $n-1$, breadth-first traversal has some more legal pairs of nodes (not mentioned in this proof) which are consecutive on disk. These may be pairs of nodes which are the south-east and north-west children of their sibling fathers. For example, nodes 6 and 8, 11 and 13, or 16 and 18 in Figure 2. They may even be the last node at one level and the first node at the next level. For example, nodes 17 and 3 in Figure 2. As has been pointed out before, in this storage order, any two nodes appearing consecutive on disk can be simultaneously black in a quadtree, with the only exception of the root and the next node in breadth-first traversal (the north-west child of the root).

Note also that Theorem 2 could be sharpened so as to quantify the difference between the searching time cost of the two storage orders. For each order (using the methodology of the proof of Theorem 2), we can find the probability of existence of every possible legal pair of nodes, multiply each such probability by 1 (2) if the corresponding two nodes appear (do not appear) consecutive on disk and sum all these quantities, reaching a formula for the respective searching time cost. Then, the fraction of Breadth-first over Preorder searching time cost could be formed.

5. CONCLUSIONS AND FUTURE RESEARCH

A new data structure, termed Dynamic Inverted Quadtree, has been introduced. This structure is a quadtree variation that can represent a set of images in secondary menory. There are other quadtree variations that are suitable for this purpose, like the Multivalued Quadtree [1] and the Fully Inverted Quadtree [2]. The former structure permits the implementation of operations needed in a GIS. The latter structure, as well as the DI-quadtree, can be used as indexes at the physical level of a pictorial database, where content oriented retrieval can be applied. The contribution of this paper is not limited to the presentation of a new structure. The set theoretic definition of region quadtrees and the branching process probabilistic model, which formalizes for the first time Shaffer's and Samet's random tree model, may prove very useful in the analysis of other quadtree based algorithms and data structures.

Based on this theoretic background we have analyzed the space efficiency of the new structure and we have shown that it exhibits certain advantages over the FI-quadtree: it is dynamic, it is more compact, reorganization is not obligatory, it can take place at any status of the structure and can be done on-line and, by using a new storage order, pattern searching is guaranteed to be done efficiently.

Future research could extend the analysis presented and compare the theoretical results with statistics of a DI-quadtree pictorial database used in real life applications. More complicated list organization methods (with varying segment sizes) that reduce the empty space in a list [4], or compress (the differences of) the image identifiers in a list [3, 14] could also be examined. In addition, systems able to solve the same problems could be designed, analyzed and compared with DI-quadtrees. For example, each node of a full class-n quadtree could be assigned a unique key (its position in breadth first traversal). Then, this key along with the associated list of image identifiers could be stored as a variable length record in a B-tree. Such a system would not occupy main memory space, but is expected to be slower in pattern matching operations.

Acknowledgements — The first author, who is a postgraduate scholar of the State Scholarship-Foundation of Greece, wishes to thank this institution for its financial assistance. Research performed by the second author was partially funded by the Institute for Systems Research (ISR), while he was with ISR and the Computer Science Department, University of Maryland, College Park, MD 20742, USA.

REFERENCES

[1] F. Arcieri and E. Nardelli. An integration approach to the management of geographical information: CARTECH. In *Proceedings of the 2nd Inter. Conf. on Systems Integration (ICSI)*, Morristown, NJ pp. 726-737 (1992).

[2] J.P. Cheiney and A. Tourir. FI-quadtree, a new data structure for content-oriented retrieval and fuzzy search. In *Proceedings of the 2nd Symposium on Spatial Databases (SSD)*, Zürich, pp. 23-32 (1991).

[3] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. on Information Theory*, **21**:194-203 (1975).

[4] C. Faloutsos and H.V. Jagadish. On B-tree indices for skewed distributions. In *Proceedings of the 18th VLDB Conf.*, Vancouver, pp. 363-374 (1992).

[5] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, **3** (3/4):231-262 (1994).

[6] C. Faloutsos, M. Ranganathan and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the ACM SIGMOD Conf.*, Mineapolis, MN, pp. 419-429 (1994).

[7] R. Laurini and D. Thomson. Fundamentals of Spatial Information Systems. *Academic Press, London* (1992).

[8] C. Puech and H.Yahia. Quadtrees, octrees, hyperoctrees: A unified analytical approach to tree data structures used in graphics, geometric modeling and image processing. In *Proceedings of the Symposium on Computational Geometry*, Baltimore, MD, pp. 272-280 (1985).

[9] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, **18** (1):35-57 (1982).

[10] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, **16** (2):187-260 (1984).

[11] H. Samet and C. A. Shaffer. A model for the analysis of neighbor finding in pointer-based quadtrees. *IEEE Trans. on PAMI*, **7** (6):717-720 (1985).

[12] M. Vassilakopoulos and Y. Manolopoulos. Analytical results on the quadtree storage-requirements. In *Proceedings of the 5th Inter. Conf. on Computer Analysis of Images and Patterns*, Budapest, pp. 41-48 (1993).

[13] J.S. Vitter and Ph. Flajolet. Average-case analysis of algorithms and data structures. In *9th chap. in Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity*, Elsevier Sc. Pub., The Netherlands (1990).

[14] J. Zobel, A. Moffat and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of the 18th VLDB Conf.*, Vancouver, pp. 352-362 (1992).