*Image Processing and Computer Vision*

*Robert Haralick Editor*

# TID—A Translation Invariant Data Structure for Storing Images

DAVID S. SCOTT and S. SITHARAMA IYENGAR

*ABSTRACT: There are a number of techniques for representing pictorial information, among them are borders, arrays, and skeletons. Quadtrees are often used to store black and white picture information. A variety of techniques have been suggested for improving quadtrees, including linear quadtrees, QMATs (quadtree medial axis transform), forests of quadtrees, etc. The major purpose of these improvements is to reduce the storage required without greatly increasing the processing costs. All of these methods suffer from the fact that the structure of the underlying quadtree can be very sensitive to the placement of the origin.*

*In this paper we discuss a translation invariant data structure (which we name TID) for storing and processing images based on the medial axis transform of the image that consists of all the maximal black squares contained in the image. We also discuss the performance of TID with other existing structures such as QMATs, forests of quadtrees, and normalized quadtrees. Some discussion on the union and intersection of images using TID is included.*

## 1. INTRODUCTION

Efficient methods for region representation are important for use in manipulating pictorial information. There are a number of techniques for representing pictorial information, among them are borders, arrays, and skeletons [31]. The quadtree has recently become an important data structure in image processing. The early history of quadtrees may be traced in papers by Alexandridis and Klinger [2], Hunter [11], Hunter and Steiglitz [12, 13], Klinger and Dyer [19], and Tanimotto and Pavlidis [36].

Methods for the region representation using quadtrees exist in the literature [5, 22, 23]. Much work has been done on quadtree properties, and algorithms for translations and manipulations have been derived by Dyer [4], Samet [24–27, 29, 30], Shneier [34], and others. The question of efficient quadtree storage was addressed by Gargantini [7], Grosky and Jain [10], Raman and Iyengar [15], Iyengar, Sadler, and Kundu [16], Jones and Iyengar [17, 18], and Samet [31]. For details on other representation on image data structure see [6, 31].

Recently Samet [31] applied the concepts of skeleton and medial axis transform to images represented by quadtrees and defined a new data structure termed QMAT. Basically, this data structure results in a partition of the image into a set of nondisjoint squares having sides of arbitrary length (but not arbitrary centers) rather than, as in the case of quadtrees, a set of disjoint squares having sides of lengths

This research was partially supported by CES-LSU.

that are powers of 2. The data structures proposed by Samet, Dyer, Rosenfeld, Jones, Iyengar, Gargantini, and others can be very sensitive to the placement of the origin.

Li, Grosky, and Jain [20] define a normalized quadtree structure with respect to translations. They have obtained an algorithm that finds the position of the quadtree requiring the minimum number of nodes. The algorithm uses a binary array representation of the image and attempts translations of magnitude power of 2 in the vertical, horizontal, and corner directions. The algorithm requires $O(2^{2n})$ space and has an execution time of $O(n.2^{2n})$($n$ is the grid resolution of the image). For a broader treatment on this see [20]. Details on space and time efficiency of virtual quadtrees can be referred to Jones and Iyengar [8].

In this paper we present a shift-invariant maximal block data structure (TID) by representing the maximal blocks by triples containing the size of the block and the coordinates of the upper left-hand corner. The TID of the image will generally have many fewer black nodes than the other corresponding structures. Empirical results confirming this are discussed in Section 5.

The remainder of the paper is organized as follows: Section 2 describes previous methods of storing images and reveals their sensitivity to the placement of the origin. Section 3 defines TIDs and presents a formal algorithm for computing them. Section 4 discusses storing and searching TIDs efficiently. Section 5 gives theoretical and computational comparisons of TIDs with other storage techniques. Section 6 concludes the paper.

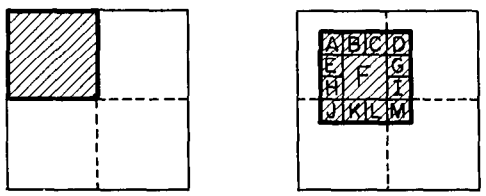



**FIGURE 1. Sample Regions**

## 2. COMPARISON OF SENSITIVITY

### 2.1 Current Representation Methods

A quadtree is a tree structure with the restriction that any node must have either four offspring (or children or descendents) or none. In a quadtree representing a picture, the root represents the whole picture. Each offspring represents one quadrant in the order northwest, northeast, southwest, southeast. In turn, their offspring each represent a subquadrant of the four quadrants and so on until every terminal node represents a region that is either all black or all white. Figures 1 and 2 show a typical picture of a simple region and its quadtree representation. In quadtrees, parents are labeled "GRAY" and leaves are either "BLACK" or "WHITE."

Various improvements to quadtrees have been suggested including forests of quadtrees [10, 17], hybrid quadtrees [15, 21], linear quadtrees [7], and optimal quadtrees for image segments [10]. All of these methods try to optimize quadtrees by removing some or all of the gray and white nodes. All of them maintain the same number of black nodes.

Recently Samet [31] presented a modification of quadtrees called QMAT (for quadtree medial axis transform). In a QMAT, black nodes in the original quadtree are allowed to expand to absorb adjacent smaller black nodes. Thus, while quadtrees decompose the image into certain disjoint squares of 2-power order, QMATs cover the image with squares of arbitrary order (but not of arbitrary center) that are not disjoint in general. Black nodes in

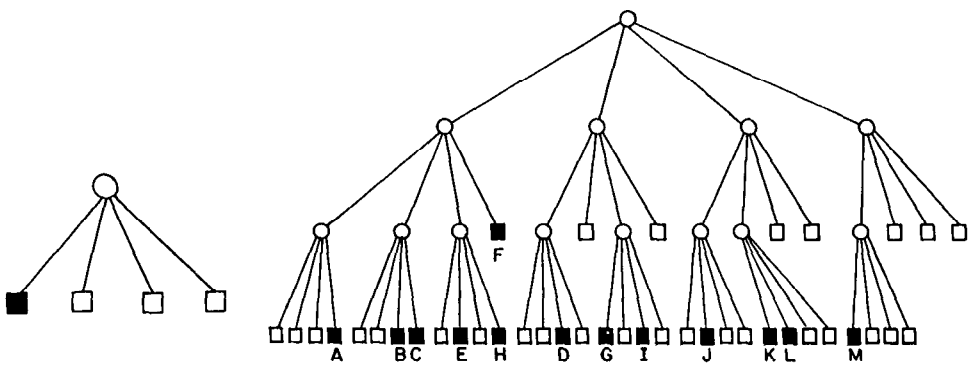



**FIGURE 2. Quadtrees for Example 1 and Example 2 of Figure 1**

QMATs are allowed to expand so that they overlap the boundary of the image. Thus QMATs can lead to a significant reduction in BLACK nodes compared to the original quadtree. Other early work on medial axis transformation includes Rosenfeld and Pfaltz's work done in 1966 [21a].

### 2.2 Sensitivity of Placement

All of the methods of representing images given suffer from sensitivity to the placement of the origin. Two images that are translations of each other can give rise to very different looking structures. We examine this phenomenon for those methods by using the example of a $2^{n-1} \times 2^{n-1}$ black square embedded in a $2^n \times 2^n$ image. In Example 1 the black square is in the upper left corner. In Example 2 it is translated down and right one pixel. Figure 1 shows Examples 1 and 2 for $n = 3$. Figure 2 gives the quadtrees for these two patterns and Table I gives the number of each kind of node. Example 1 in Table I is constant for all values of $n$. The derivation of the entries in Example 2 for arbitrary $n$ is given in [32].

*Observation 1*: The quadtree for Example 2 grows exponentially in $n$ and the quadtree for Example 1 has 5 nodes independent of $n$. How do the various representation schemes apply to Example 2? Most of the schemes eliminate most or all of the pointers and white nodes and do nothing to black nodes. Thus linear quadtrees, compact quadtrees, hybrid quadtrees, and forests of quadtrees are all inevitably forced to store $3(2^n - n) - 2$ black nodes plus perhaps some others. The scheme capable of eliminating black nodes is given in [31].

*Observation 2*: The QMAT for Example 2 has an interesting structure. Most of the image is covered by only four nodes but a sequence of decreasing sized nodes is needed to cover the rest. Figure 3 shows the 172 black blocks for Example 2 with $n = 6$. The four blocks labeled A expand to cover most of the square but the blocks labeled B are needed to cover the rest.

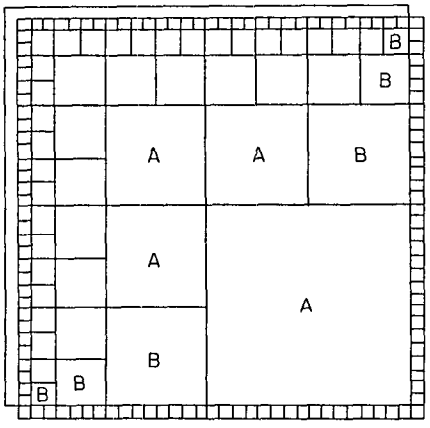



**FIGURE 3. The 172 Black Blocks for Example 2 with $n = 6$**

**TABLE I. Number of Each Kind of Node in the Quadtrees for $n = 3$**

| | Example 1 | Example 2 | For arbitrary $n$ |
|---|---|---|---|
| Grey node | 1 | 13 | $2^{n+1} - 3$ |
| White node | 3 | 27 | $3(2^n + n - 2)$ |
| Black node | 1 | 13 | $3(2^n - n) - 2$ |

Based on the above observation, we can state the following theorem.

THEOREM 1
*For $n \geq 5$ the QMAT for Example 2 has $2n - 2$ black nodes, $4n - 6$ white nodes, and $2n - 3$ gray nodes.*

PROOF
See [32].

*Observation 3*: The shift sensitivity of the image data structures (such as quadtrees, compact quadtrees, and QMATs) derives from the fact that the positions of the maximal blocks are not explicitly represented in the data structure. Instead, these positions are determined by the paths leading to them from the root of the tree. Thus, when the image is shifted, the maximal blocks are formed in a different way.

Sensitivity to the placement of the origin is particularly annoying when translating images (e.g., when several images are combined). As the example above shows, even small translations can make enormous changes in the underlying representation. The possibility for black nodes to overlap the boundary in QMATs creates a further obstacle to correctly combining several QMATs. In the next section we introduce a new data structure for storing images that is translation invariant.

## 3. TID—A TRANSLATION INVARIANT DATA STRUCTURE

### 3.1 The Medial Axis Transform

The maxnorm (or infinity norm) of a point $(a, b)$ is $\max(\|a\|, \|b\|)$. The distance between two points $(a, b)$ and $(c, d)$ is

$$D((a, b), (c, d)) = \max(\|a - c\|, \|b - d\|).$$

The set of all points $B$ which are a distance $\alpha$ from a fixed point $A$ is a square of size $2\alpha$ centered on $A$.

In any image, a maximal black square is any

square of black pixels that is not contained in any larger square of black pixels. The medial axis transform with respect to the maxnorm is the set of all maximal black squares contained in the image. This concept was exploited by Samet in deriving QMATs. The medial axis transform (MAT) of an image is clearly translation invariant since it only depends on the intrinsic geometry of the image. In this section we will investigate various aspects of MATs. In particular, Section 3.4 discusses the desirability of eliminating redundant squares.

### 3.2 Computing Distances

Before determining the maximal squares in an image, it is first necessary to compute the distance from each black pixel to the nearest white pixel (or boundary). $D$ can be computed in linear time (in the number of pixels) using the algorithm given by Borgefors [3].

### 3.3 Locating Maximal Squares

*Definition of a Maximal Black Square*: A maximal square is a black square of pixels that is not contained in any large square.

Let $D(i, j)$ be the distance from $(i, j)$ to the nearest white pixel. A maximal square is a black square of pixels that is not contained in any larger black square. The square centered on $(i, j)$ will be the largest black square centered on $(i, j)$ (which will always be of odd order with side $s = (2 * D(i, j) - 1)$). A constant 2-square will be a $2 \times 2$ square of pixels that all have the same $D$ value. The square centered on a constant 2-square will be the largest black square centered on the 2-square (which will have even order with side equal to twice the constant $D$ value). Two pixels will be considered adjacent if they share a common side or corner. Two adjacent pixels will be called neighbors. The key result for locating maximal squares is stated in the following theorem.

THEOREM 2

*A black square is maximal if and only if either*: (1) *It is centered on a constant* 2*-square or* (2) *It is centered on* $(i, j)$ *and both* (*a*) $D(i, j)$ *is a local maximum of* $D$ *and* (*b*) $(i, j)$ *is not part of a constant* 2*-square.*

PROOF

($\rightarrow$) Let $Q$ be a maximal square. Suppose $Q$ has even order and let $T$ be the 2-square on which $Q$ is centered. Suppose $T$ is not constant. Let $D(i, j) > D(k, m)$ for two pixels in $T$. Then the odd ordered square centered on $(i, j)$ strictly contains $Q$ which is a contradiction. So $T$ must be a constant. This completes Part 1 of the theorem.

Suppose $Q$ has odd order with center $(i, j)$. Suppose $(k, m)$ is adjacent to $(i, j)$ with $D(k, m) > D(i, j)$. Then the square centered on $(k, m)$ will strictly contain $Q$, which is a contradiction. This is Part 2a of the theorem. Suppose $(i, j)$ is part of the constant 2-square $T$. Then the square centered on $T$ strictly contains $Q$, which is a contradiction. This is Part 2b of the theorem. This completes ($\rightarrow$).

($\leftarrow$) Let $Q$ be the square centered on the constant 2-square $T$ and suppose $Q$ is strictly contained in some larger square $S$. Let $(i, j) \in T$ and $(k, m) \in T$ be such that $(i, j)$ is closer to the center of $S$ than $(k, m)$. Then the square centered on $(i, j)$ is contained in $S$ and contains no boundary squares of $S$, which contradicts the definition of $D(i, j)$. Thus $Q$ must be maximal. Suppose $Q$ is centered on $(i, j)$ and both Parts 2a and 2b hold and suppose $Q$ is strictly contained in a larger black square $S$.

Suppose $S$ is of odd order centered on $(k, m)$. By symmetry we may assume either (a) $i = k$ and $j < m$ or (b) $i < k$ and $j < m$. In Case (a) $D(i, j + 1) = D(i, j) + 1$. In Case (b) $D(i + 1, j + 1) = D(i, j) + 1$. Both possibilities contradict Part 2a and so $S$ may not be of odd order.

Suppose $S$ is of even order with central 2-square $T$. Let $(k, m)$ be the pixel in $T$ closest to $(i, j)$. If $(i, j) \neq (k, m)$, then the odd square centered on $(k, m)$ strictly contains $Q$. This is the previous case which contradicts Part 2a.

Finally suppose $(i, j) = (k, m)$. If $(n, p)$ is some pixel in $T$, then $D(n, p)$ cannot be greater than $D(i, j)$ by Part 2a and if $D(n, p)$ is less than $D(i, j)$, then $Q$ is not contained in $S$. Thus $T$ is a constant 2-square, which contradicts Part 2b. Hence $Q$ is maximal. □

The characterization of maximal squares in Theorem 2 is purely local and all maximal squares can be identified in one pass through the pixels. Thus the maximal squares can be located in linear time. In fact, the squares can be identified during the second pass of the Borgefors algorithm [3] and so no additional pass is needed.

### 3.4 Eliminating Redundant Maximal Squares

Not all maximal squares may be needed to cover an image. Figure 4 displays a black rectangle that is the union of two squares. However, the image contains six maximal squares (each centered on a 3). It is clearly desirable to eliminate unnecessary maximal squares. The squares at the ends are needed and all of the others are not. In this case, a unique pair of maximal squares covers the image. If we make the rectangle 11 wide instead of 10, then one additional square is needed. Both ends are still required but any one of the five interior squares could be used. (See Figure 5.)

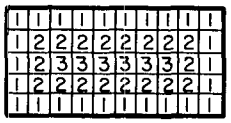


**FIGURE 4. A Black Rectangle That is the Union of Two Squares**

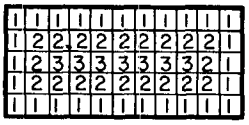


**FIGURE 5. Expansion of the Rectangle in Figure 4**

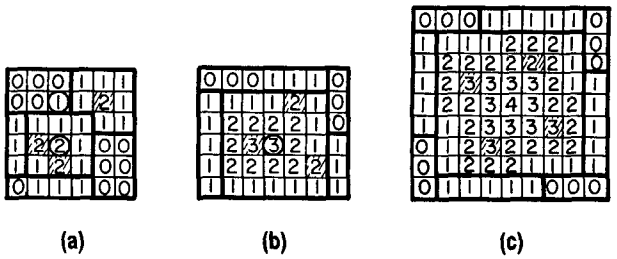



**FIGURE 6. Examples to Show That There is No Fast Way of Determining Whether a Particular Maximal Square is Covered by Three or Four Maximal Squares**

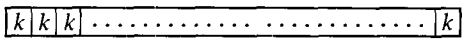
Unfortunately, there appears to be no fast (local) way of determining whether a particular maximal square is covered by three or four maximal squares. Figure 6a–c gives some examples illustrating this point. In each case, the maximal square centered on the circled pixel is covered by the maximal squares centered on the darkened pixels.

In general, there are complex dependencies among the maximal squares that are expensive to compute and difficult to analyze, which makes it difficult to determine an optimal subset to delete.

For these reasons we will only consider eliminating maximal squares that are covered by only two other maximal squares. These seem by far to be the most common kind of dependency and more importantly, such a dependency has a local characterization. Namely, let $Q$ be a maximal square centered on $(i, j)$. $Q$ can be covered by two other squares if and only if one of the following holds: (a) $D(i, j-1) = D(i, j) = D(i, j+1) > 1$, (b) $D(i-1, j) = D(i, j) = D(i+1, j) > 1$.

This means that such redundant squares are always signaled by a consecutive sequence of pixels (either horizontally or vertically) with constant $D$ value. Only local maxima of $D$ need be considered for elimination, so we assume all of the pixels are local maxima except possibly the ends. Let the $D$ value be $k$ and let $n$ be the maximum number of consecutive pixels with constant $D$ value.

$n$ pixels




How many of the squares are redundant? Two squares cover everything in between provided that at most $2k-2$ centers lie in between. Thus, to determine which squares to include, start at one end, include the end square, delete the $2k-2$ square, include the next, delete the next $2k-2$, include the next, and so on until the end is reached. Always include the other end. If the endpoints are not local maxima, then they are "included" in whatever large square contains them.

Figure 7 shows an example with $k = 2$ and $n = 8$. Note that both endpoints are not local maxima. If the algorithm starts at the left, and the circled centers are kept, then two squares are deleted for each one kept because $2k-2$ equals 2. The endpoints are covered by the circled 3s. Obviously the squares that are kept may depend on whether the algorithm starts at the left or right end but the number of squares deleted may be the same.

Assuming that the algorithm always goes left to right (and top to bottom), the results of the algorithm on one sequence are fixed. The only remaining question is what happens when a horizontal and a vertical sequence intersect. Does it matter which sequence is processed first? The answer is yes. Processing in the wrong order may cause retention of one more square than necessary. In Figure 8, if the row is processed first, then the six circled centers are kept. If the column is processed first, then the central square is deleted before the row is processed. This breaks the row into two separate pieces which are processed separately. This results in the marked square being deleted.

This phenomenon has nothing to do with which sequence is longer. It can occur only when the central square would be deleted in both directions. This problem has a simple solution. Process all horizontal sequences first. Whenever a square is about to be deleted check to see if the center of the square is part of a vertical sequence. If it is, then do not delete the sequence and start the delete count over (i.e., delete to the next $2k-2$ squares). The vertical pass is unaffected.

The question of even order maximal squares remains. As shown in Theorem 2, an even order maximal square is characterized as centered on a constant 2 square. As for the odd order maximal squares, there are a variety of ways an even order maximal square might be redundant. We will only consider two of them.

THEOREM 3

*Let $T$ be a constant 2-square. The maximal square $Q$ centered on $T$ is redundant if either* (1) *or* (2) *holds:*
(1) *None of the four pixels in $T$ are local maxima of $D$.*
(2) *There exist two other constant 2-squares $R$ and $S$ such that $R \neq T \neq S$ and $T \subseteq R \cup S$.*

PROOF

If a pixel $P$ is not a local maxima of $D$, then the square centerd on $P$ is contained in some larger odd order square. If all four of the pixels in $T$ are not local maxima, then $Q$ can be covered with four larger odd order squares. Hence $Q$ is redundant.

The only way 2 can hold is if there is a $2 \times 8$ rectangle of constant $D$ values with $T$ being the central square. Then $Q$ is covered by the squares centered on $R$ and $S$, so $Q$ is redundant.

Theorems 2 and 3 together indicate that we need only concern ourselves with pixels that are local maxima of $D$. Redundant squares in sequences of equal 2-squares can be handled in the same way as for odd order sequences.

The algorithms are given in the Appendix.

*Fact 1*: The TID of an image is a subset of the MAT of the image. In particular, most of the redundant maximal squares (all the maximal squares that are covered by two others) can be computed in time which is linear in the number of pixels. Proof of this fact is given in [32].

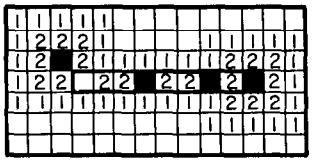


**FIGURE 7. An Example with $k = 2$ and $n = 8$**

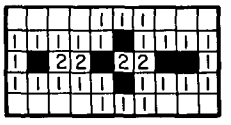


**FIGURE 8. An Example of Row Processing**

## 4. STORING AND SEARCHING A TID

Each maximal square in a TID is characterized by three numbers (two to specify a location and one to specify size). Using the center of the square for location does not work well for even order squares, so we will use the coordinates of the upper left corner of the square and its size as the three parameters. Thus each square in a TID has the representation $(i, j, s)$.

Unfortunately, unlike linear quadtrees, such triples of numbers do not have a natural linear ordering. The best thing that can be done is to choose some priority order for the coordinates and then order them lexicographically. We will assume that $i$ and $j$ are sorted in increasing order but for reasons which will become clear shortly, we will assume that $s$ is sorted in decreasing order. By symmetry we may assume that $i$ is ordered before $j$. Thus the question is which of the three possible ordering plans: (a) $(i, j, s)$, (b) $(i, s, j)$, or (c) $(s, i, j)$, is the best. There is no definitive answer to this question. It depends on whether storage or access is of primary concern.

Storage is conserved when the primary subdivisions are large since the value of the primary variable will only be stored once. On this grounds, Plan (a) can be eliminated since there can be at most one maximal square with corner $(i, j)$ and so no savings can be obtained at the second division compared to Plan (b) or Plan (c). Plan (b) may be better since there may be several squares of the same size with the same $i$ value.

For storage purposes the competition is between Plan (b) and Plan (c). For most images Plan (c) is superior since there will be many small squares around and so the subsets of size 1, 2, and 3 will be quite large allowing for a much greater space savings than can be obtained by Plan (b).

For searching it is important to shorten the length of the search whenever possible by skipping to the beginning of the next primary or secondary classification. The following discussion assumes that the purpose of the search is to decide whether pixel $(k, m)$ is black, that is, whether $(k, m)$ is contained in some square in the TID.

On this basis Plan (c) can be ruled out since it is impossible to determine anything given just $s$. On the other hand with Plan (a) or Plan (b) we can stop the search as soon as $k < i$. The question of whether Plan (a) or Plan (b) allows more skipping of squares remains. For fixed $i$, Plan (a) skips all squares with $j > m$. Plan (b) skips all squares with $s < k - i$. Which set is larger? This obviously depends on the image. For the purpose of analysis we make the fol-

lowing assumptions: (1) Every value of $j$ is equally likely. (2) All possible values of $s$ (i.e., 1 to number-$i$ + 1) are equally likely.

Assumption 1 is true only for "random" squares only if they have size 1. For larger squares the distribution is skewed, favoring smaller values of $j$. Assumption 2 is even less reasonable. In most images there will be many more small squares than large squares.

For fixed $i$, Plan (a) skips all squares $(1, j, s)$ with $1 = i$ and $j > m$. By Assumption 1 this will be about half the squares with $1 = i$ for a random $(k, m)$. Plan (b) will skip all squares $(1, j, s)$ with $1 = i$ and $1 + s < k$. By Assumption 2 this will be about half the squares with $1 = i$. Thus by analysis the two plans are about the same. However, both biases in the assumption favor Plan (b), particularly the second one. Thus Plan (b) is better.

Unfortunately, only some constant fraction of the squares can be skipped and so the search time is still linear in the number of squares.

### 4.1 Translation, Rotation, and Union of TIDs

A TID is made up of maximal squares. Each square is represented as a triple $(i, j, s)$, where $(i, j)$ is the northwest corner of the square and $s$ is the length of the side. Thus a TID is just a list of such tuples. Translations and rotations applied to the image are just simple functions of these tuples. To translate a triple $(i, j, s)$ by $I$ units right and $J$ units up yields

$$T_{ij}(i, j, s) = (i + I, j + J, s).$$

Rotation by $\pi/2$ is only slightly more complicated due to the fact that the NW corner of the square changes upon rotation. The $\pi/2$ rotation around the origin is

$$R(i, j, s) = (-j, i + s, s)$$

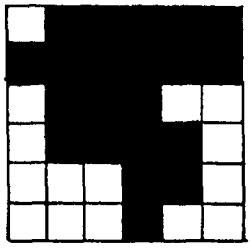
Rotations around other coordinates can be obtained by composing $R$ with the appropriate translations.

Union of TIDs is straightforward—simply take the union of the two lists. The resulting list will be the TID of the combined images whenever the images do not overlap. Two problems can occur when the two images overlap. A square in one TID may be contained in a square from the other TID and thus be redundant, or several squares from both TIDs could be combined to form a larger square. The first problem can be easily checked for. The second problem is harder—there appears to be no better solution for reforming the array of pixels and recomputing the TID. On the other hand the combined lists are unlikely to contain many such larger squares (if any) and thus should be an adequate representation of the image.

We analyze the union of two regions arising (or suitably projected) from the superpositioning of the images using TID as described in our other paper [33]. The reader is referred to [33] for the basic idea behind manipulation algorithms on TID and its relation to other representations.

## 5. THEORETICAL AND COMPUTATIONAL COMPARISONS OF TIDs WITH OTHER DATA STRUCTURES

The purpose of this section is to compare the storage requirements and preprocessing costs of TIDs with those of quadtrees, linear quadtrees, forests of quadtrees, and QMATs.



(a)

```
0 1 1 1 1 1
1 2 2 2 1 1
0 2 2 3 0 0
0 1 1 3 1 0
0 0 0 2 1 0
0 0 0 1 0 0
```

(b)

```
0 1 1 1 1 1
1 1 2 1 1 1
0 1 2 1 0 0
0 1 1 1 1 0
0 0 0 1 1 0
0 0 0 1 0 0
```

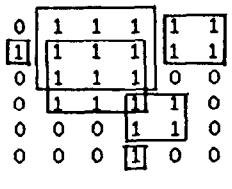
(c)

```
X 0 0 0 1 1
1 0 1 0 1 1
X 0 1 0 X X
X 0 0 0 1 X
X X X 1 1 X
X X X 1 X X
```

(d)

```
X 0 0 0 3 2
1 0 1 0 2 2
X 0 1 0 X X
X 0 0 3 2 X
X X X 2 2 X
X X X 1 X X
```

(e)



(f)

**FIGURE 9. Sample Region and Its Corresponding TID Representation**

It is well known that the storage requirements for these modified quadtree data structures are very shift sensitive. A rigorous analysis is given by Iyengar and Lewis [14]. For the present discussion, consider the image in Figure 9a. This region must be embedded in a square of size $2^3 \times 2^3$. There are nine possible locations (shown in Figure 10a–i) for the embedded region within the square. Maximal square characterization of Figure 9a using TID structure is described in Table I. Table II summarizes the best, worst, and average performance for the various locations.

On the average, TID provided a 56 percent space reduction over linear quadtrees and an 86 percent reduction when compared to other data structures. Reductions may not always be this spectacular.

THEOREM 4

*The number of nodes required to store a TID does not exceed the minimum number required by optimally located quadtrees, linear quadtrees, forests of quadtrees, or QMATs.*

PROOF

The number of black nodes are identical for quadtrees, linear quadtrees, and forests of quadtrees. Each such black node is a subset of a maximal block. This also holds for QMATs, since there are restrictions on the merging allowable under QMATs. Therefore, at worst, there is one-to-one correspondence between black nodes and TID nodes. (Notice that quadtrees, forests of quadtrees, and QMATs must also store white nodes.)

This space savings is not without costs. Scott and Iyengar [20] note that the time required to construct a TID is $O(rc \log(\min(r, c)))$, where $r$ and $c$ are the number of rows and columns in the embedded region.

Suppose $2^{r-1} < \max(r, c) \leq 2^r$ and let $s = 2^r$. The time complexity to construct a TID is at worst $O(s^2 \log s)$, and may be much less since the embedded region may not fill the $s \times s$ square. By comparison, the time complexity to construct a modified quadtree is $O(s^2)$. This slight reduction in time may be more than offset by the complexity of the resulting tree if a poor location is chosen.

**TABLE II. Maximal Square Characterization of Figure 9a Using TID Structure**

| Row | Column | Size |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 5 | 2 |
| 2 | 1 | 1 |
| 2 | 2 | 3 |
| 4 | 4 | 2 |
| 6 | 4 | 1 |

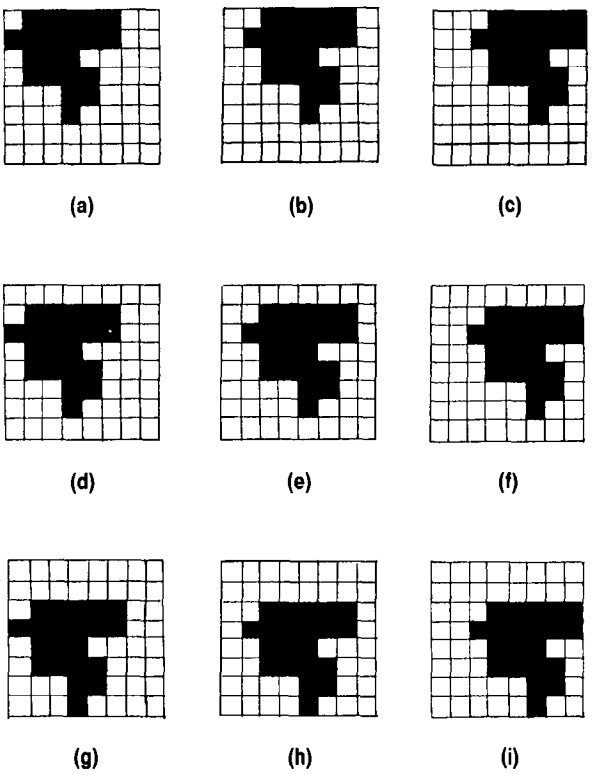



**FIGURE 10. Possible Placements of the Sample Region of Figure 9 Inside a Square of Size $2^3 \times 2^3$**

Is TID still advantageous if a search is made for a good location? The costs associated with such a search must be considered. Grosky and Jain [10] define the normalized quadtree of an image as its quadtree constructed for the location that results in the minimum number of nodes. If the number of nodes are minimized at more than one location, Grosky and Jain [10] use the northern- and western-most of these locations. The extension of their definition to normalize linear quadtrees, forests, and QMATs is obvious.

Grosky and Jain notice that the normalized quadtree may require a $2^{k+1} \times 2^{k+1}$ square. We now extend their result to other data structures.

THEOREM 5

*If an image can be embedded in a $2^k \times 2^k$ square, the maximum square needed to normalize its data structure is: $2^{k+1} \times 2^{k+1}$ for quadtrees, linear quadtrees, and QMAT; and $2^{k+2} \times 2^{k+2}$ for forests of quadtrees.*

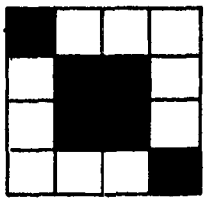


FIGURE 11. An Example Using a $2^{k+1} \times 2^{k+1}$ Figure For Optimal Positioning

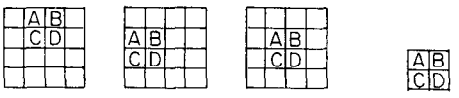



FIGURE 12. Subimages of Arbitrary Complexity

PROOF

Figure 11 is an example where it is necessary to use a $2^{k+1} \times 2^{k+1}$ square for optimal positioning. If an image requires a $2^{k+2} \times 2^{k+2}$ square, the optimal position cannot be in a principal quadrant (i.e., quadrants from the root). Figure 12a–c details the possible cases, where *A*, *B*, *C*, and *D* represent subimages of arbitrary complexity. Each of these cases is worse for the quadtree or QMAT, and no better for a linear quadtree than the placement of Figure 12d in a $2^{k+1} \times 2^{k+1}$ square.

Figure 12c, however, may represent a better placement for the forests of quadtrees data structure if two of the subimages are good black. This will be the case if both remaining subimages are white or if at least one of them has a good white gray root. □

*Fact 2:* Theorem 5 defines the size of the area that must be searched. TIDs on the other hand need only the $r \times c$ enclosing rectangle ($r \times c \leq 2^k \times 2^k$) since no location search is necessary.

For the example in Figure 10a, the optimal locations for most of the modified quadtrees occur within the $2^3 \times 2^3$ square of Figure 11a–i. (See Table III.) The exception is the forests of quadtrees, which is normalized by a horizontal translation at 3 pixels. The performance of normalized data structures is compared in Table IV.

To normalize the image representation, each of the $4^k$ ($=s^2$) possible locations ($4^{k+2}$ for forests of quadtrees) must be examined to determine its suitability. The cost of constructing any of these modified quadtrees is $O(s^2)$. If the suitability of any location is assessed by actually constructing the data structure, then the normalization cost is $O(s^4)$.

Some reduction is possible for quadtrees and linear quadtrees. Grosky and Jain [10] give an $O(s^2 \log s)$ algorithm to find the optimal location. This algorithm may be adopted to linear quadtrees with only a slight increase in the proportionality constant. The algorithm is based on a process of merging nodes from the bottom up while keeping track of the potential number of leaves. It does not extend to forests of quadtrees or QMATs. For instance, if a forest of good white gray nodes is merged with at least two good black nodes, all of the descendant gray nodes and white leaves would need to be counted, whereas most of them would otherwise be discarded.

We can define the cost of preprocessing as the total number of operations required to construct the data structure. If translation invariance is achieved by searching for the optimal placement, then the cost of the search much be considered in the preprocessing costs. Table V compares the total preprocessing costs for these data structures.

The Grosky and Jain algorithm finds the optimal

TABLE III. Number of Nodes Required to Represent Figure 10a–i

| Location | Quadtree | Linear Quadtrees | Forests of Quadtrees | QMAT | TID |
|---|---|---|---|---|---|
| Best | 33 | 12 | 26 | 33 | 6 |
| Worst | 57 | 18 | 58 | 57 | 6 |
| Average case | 44 | 13 | 42 | 44 | 6 |

TABLE IV. Number of Nodes Required for Normalized Data Structures

| Data Structures | Total Number of Nodes |
|---|---|
| Normalized quadtree | 33 |
| Normalized linear quadtree | 12 |
| Normalized forest of quadtrees | 24 |
| Normalized QMAT | 33 |
| TID | 6 |

TABLE V. Preprocessing Costs for Normalized Data Structures

| Data Structure | Cost of Preprocessing |
|---|---|
| TID | $O(S^2 \log S)$ |
| Normalized quadtree | $O(S^2 \log S)$ |
| Normalized linear quadtree | $O(S^2 \log S)$ |
| Normalized forest of quadtrees | $O(S^4)$ |
| Normalized QMAT | $O(S^4)$ |

location without actually constructing the data structure. Once the optimal location is found, the data structure must still be constructed, although the total time required is dominated by the search time.

As was noted previously, search time to normalize a quadtree is less than that for a linear quadtree. TID requires no normalizing search, so the only cost is the actual construction cost. Therefore, we expect the TID to be slightly more cost efficient and storage efficient than the other structures.

## 6. CONCLUDING REMARKS

(1) The greatest advantage of TIDs over other methods of storing images is that TIDs are translation invariant. This is particularly important if several images are combined into one composite. Another interesting feature of TIDs is the fact that the image itself need not be a square of 2-power order. A square of any order or even any rectangular image can be represented without having to embed it in a square of 2-power order.

For example, if a black square of order $(2^k + 1) \times (2^k + 1)$ needs to be stored, the TID would just be the black square. To store it as a quadtree requires that it be embedded in a $2^{k+1} \times 2^{k+1}$ square. The best embedding would require $2^{k+1} + 2$ black leaves.

(2) The second advantage of a TID is that the number of black squares stored may be significantly less than the number in the corresponding quadtree. This is important in such tasks as drawing the image where the time required will be proportional to the number of squares. For example, Tamminen [35] quotes 5198 black leaves for the quadtree encoding the circle inscribed in a $2^{10} \times 2^{10}$ square. The corresponding TID has 601 black squares, an 83 percent reduction.

Scott and Iyengar [32] note that the time required to construct a TID is $O(rc \log(\min(r, c)))$, where $r$ and $c$ are the number of rows and columns in the embedded region. Suppose $2^{r-1} < \max(r, c) \leq 2^r$ and let $s = 2$. The time complexity to construct a modified quadtree is $O(s^2)$. This slight reduction in time may be more than offset by the complexity of the resulting tree if a poor location is chosen. The time complexity to construct a TID is at worst $O(s^2 \log s)$, and may be much less, since the embedded region may not fill the $s \times s$ square. By comparison, the time complexity to construct a modified quadtree is $O(s^2)$. This slight reduction in time may be more than offset by the complexity of the resulting tree if a poor location is chosen. (TIDSOFT—A software to manipulate different image settings implemented on a VAX-11/780 is available on request.)

In principle the TID for a $2^n \times 2^n$ image requires $3n$ bits for each maximal square. In practice this can be reduced by techniques described in Section 3.4. But the savings is never more than a factor of 3 (the last parameter at least must always be stored for every square). Thus the circle example from Section 4 could not be stored in less than 6010 bits. (In fact, this particular example would take about 15000 bits since there are only four squares of each size, which is not much less than the 17,905 quoted by Tamminen [35].)

## APPENDIX

```
procedure deletesquare (a, numrow,
  numcol)
(* delete unneeded local maxima and *)
(* mark essential constant 2-squares *)
begin
  deleterows (a, numrow, numcol);
  deletecols (a, numrow, numcol);
end;

procedure deleterows (a, numrow,
  numcol);
(* delete unneeded local maxima by
  scanning rows *)
(* count counts the number of
  consecutive *)
(* deleted squares *)
begin
  for i := 2 to numrow-1 do
  begin
     d := 0;
     for k := 1 to numcol-1 do
     if a[i,k].horiz <> 1 or
       (a[i-1,k].dist = a[i,k].dist
         and
       a[i+1,k].dist = a[i,k].dist)
         then
     begin
       d := a[i,k].dist;
       count := 0;
       maxcount := 2 * (d-1);
     end
     else
     if a[i,k].dist <> d then
     begin
         twosquare(i,k,no);
       if a[i,k].horiz <> 3 then
       twosquare(i-1,k);
       d := a[i,k].dist;
       count := 0;
       maxcount := 2 * (d-1)
     end
     else
     if count = maxcount then
```

```
      begin
        twosquare(i,k);
        if a[i,k].horiz <> 3 then
          twosquare(i-1,k);
        count := 0;
      end
      else
      if a[i,k].dist = d then
      begin
        count := count + 1;
        a[i,k].horiz := 0;
      end
    end
end;

procedure deletecols (a, numrow,
  numcol);
(* delete unneeded local maxima by
  scanning cols *)
begin
  for i := 2 to numcol-1 do
  begin
      d := 0;
      for k := 1 to numrow-1 do
      if a[i,k].horiz <> 0 then
      begin
        d := a[i,k].dist;
        count := 0;
        maxcount := 2 * (d-1);
      end
      else
      if a[i,k].dist <> d then
      begin
        twosquare(i,k,no);
        if a[i,k].horiz <> 3 then
          twosquare(i-1,k,no);
        d := a[i,k].dist;
        count := 0;
        maxcount := 2 * (d-1)
      end
      else
      if count = maxcount then
      begin
        twosquare(i,k);
        if a[i,k].horiz <> 3 then
          twosquare(i,k-1,no);
        count := 0;
      end
      else
      if a[i+1,k].dist = d then
      begin
        count := count + 1;
        a[i,k].horiz := 0;
      end
    end
end;

procedure twosquare (i,j);
(* check is a[i,j] is the NW *)
(* corner of a constant 2-square *)
begin
  d := a[i,j].dist;
  if a[i+1,j].dist = d and
  a[i,j+1].dist = d and
  a[i+1,j+1].dist = d then
 begin
  a[i,j].horiz := 3;
  a[i,j+1].horiz := 2;
  a[i+1,j].horiz := 2;
  a[i+1,j+1].horiz := 2;
 end
end;
```

*Acknowledgments.* We thank Professor Hanan Samet for his constructive comments and criticism of our paper. The comments of the referees, the technical editor, Professor Haralick, Steven L. Tanimotto, and our graduate students Nick Lakahni, Nancy Gauthier, Jerry W. Lewis, and S.V.N. Rao on an earlier version of this paper are greatly appreciated.

**REFERENCES**

Note: Reference [9] is not cited in the text.

1. Abel, D.J., and Smith, J.L. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Comput. Graphics Image Processing 24*, 1 (Oct. 1983), 1–13.
2. Alexandridis, N., and Klinger, A. Picture decomposition, tree data-structures and identifying directional symmetries as node combinations. *Comput. Graphics Image Processing 8*, 4 (1978), 43–47.
3. Borgefors, S. *Comput. Graphics Image Processing* 27, 3 (1983).
4. Dyer, C.R. Computing the Euler number of an image from its quadtree. *Comput. Graphics Image Processing 13*, 3 (1980), 279–276.
5. Dyer, C.R., Rosenfeld, A., and Samet, H. Region representation: Boundary codes for quadtrees. *Commun. ACM 23*, 3 (Mar. 1980), 171–179.
6. Ferrari, L., Sankar, P.V., and Sklansky, J. Minimal rectangular partitions of digitized blobs. *Comput. Vision, Graphics Image Processing 28*, (Oct. 1984), 58–71.
7. Gargantini, I. An effective way to represent quadtrees. *Commun. ACM 25*, 12 (Dec. 1982), 905–910.
8. Gauthier, N.K., Iyengar, S.S., Lakhani, N.B., and Manohar, M. Space and time efficiency of the forest of quadtrees representation. *J. Image Vision Comput. 3*, 2 (May 1985), 63–70.
9. Gauthier, N.K., Iyengar, S.S., Scott, D.S., Lakhani, N.B., and Lewis, J. Performance analysis of TID. In *Proceedings of IEEE Computer Vision and Pattern Recognition Conference*, (San Francisco, Calif., June 9–13, 1985), 416–418.
10. Grosky, W.I., and Jain, R. Optimal quadtrees for image segments. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-5*, 1 (1983), 77–83.
11. Hunter, G.M. Efficient computation and data structures for graphics. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Sciences, Princeton Univ., Princeton, N.J., 1978.
12. Hunter, G.M., and Steiglitz, K. Operations on images using quadtrees. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-1*, 2 (Apr. 1979).
13. Hunter, G.M., and Steiglitz, K. Linear transformation of pictures represented by quadtrees. *Comput. Graphics Image Processing 10*, 3 (July 1979), 289–296.

14. Iyengar, S.S., and Lewis, J.W. Translation properties and the space efficiency of modified quadtree data structure. Tech. Rep. 84-031, Louisiana State Univ., Baton Rouge, 1984.
15. Iyengar, S., and Raman, V. Properties of the hybrid quadtrees. In *Proceedings of the 7th International Conference on Pattern Recognition.* (Montreal, Quebec, Canada, July 30–Aug. 2, 1984), 824–827.
16. Iyengar, S.S., Sadler, T., and Kundu, S. A technique for representing a tree structure with predicates by a forest data structure. Tech. Rep. 84-029, Dept. of Computer Science, Louisiana State Univ., Baton Rouge, 1984.
17. Jones, L., and Iyengar, S.S. Representation of regions as a forest of quadtrees. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, (Dallas, Tex., 1981), 57–59.
18. Jones, L., and Iyengar, S.S. Space and time efficient virtual quadtrees. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-6*, 2 (Mar. 1984), 244–247.
19. Klinger, A., and Dyer, C.R. Experiments in picture representation using regular decomposition. *Comput. Graphics Image Processing 5*, 1 (1976), 68–105.
20. Li, M., Grosky, W.I., and Jain, R. Normalized quadtree with respect to translations. *Comput. Graphics Image Processing 20*, 1 (Sept.), 72–81.
21. Raman, V., Iyengar, S., and Kundu, S. An optimized quadtree structure for pictorial data representation using top and bottom compaction techniques. In *Proceedings of the IEEE Systems, Man, and Cybernetics Conference*, (Bombay, India, Dec. 1983), 771–776.

21a. Rosenfeld, A., and Pfaltz, J.L. Sequential operators in digital picture processing. *J. ACM 13*, (1966), 471–494.

22. Samet, H. Region representation: Quadtrees from binary arrays. *Comput. Graphics Image Processing 18*, 1 (1980), 88–93.
23. Samet, H. Region representation: Quadtrees from boundary codes. *Commun. ACM 23*, 3 (Mar. 1980), 163–170.
24. Samet, H. An algorithm for converting rasters to quadtrees. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-3*, (Jan. 1981), 93–95.
25. Samet, H. Connected component labeling using quadtrees. *J. ACM 28*, (July 1981), 487–501.
26. Samet, H. Computing perimeters of images represented by quadtrees. *IEEE Trans. Pattern Analysis Machine Intell. PAMI-3*, 6 (1981), 683–687.
27. Samet, H. Neighbor finding techniques for images represented by quadtrees. *Comput. Graphics Image Processing 18*, 1 (Jan. 1982), 37–57.
28. Samet, H. Data structures for quadtree approximation and compression. Computer Science TR-1209, Univ. of Maryland, College Park, Md., Aug. 1982.
29. Samet, H. Distance transform for images represented by quadtrees. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-4*, 3 (1982), 298–303.
30. Samet, H. Algorithms for the conversion of quadtrees to rasters. *Comput. Graphics Image Processing 26*, 1 (Apr. 1984), 1–16.
31. Samet, H. A quadtree medial axis transform. *Commun. ACM 25*, 9 (Sept. 1983), 680–693.
32. Scott, D., and Iyengar, S.S. TID—A translation invariant data structure for storing images. Tech. Rep. 84-027, Dept. of Computer Science, Univ. Texas at Austin, 1984.
33. Scott, D., and Iyengar, S. A new data structure for efficient storing of images. *Pattern Recognition Lett.*, 3 (1985), 211–214.
34. Shneier, M. Path-length distances for quadtrees. *Inform. Sci. 23*, 1 (1981), 49–67.
35. Tamminen, M. Comment on quad- and octtrees. *Commun. ACM 27*, 3 (Mar. 1984), 248–249.
36. Tanimoto, S., and Pavlidis, T. A hierarchical data structure for picture processing. *Comput. Graphics Image Processing 4*, 2 (June 1975), 104–119.



**CR Categories and Subject Descriptors:** [**Data**]: Data Structures—*trees*; I.2.10[**Artificial Intelligence**]: Vision and Scene Understanding—*representations, data structures and transforms*

**General Terms:** Algorithms, Theory

**Additional Key Words and Phrases:** translation invariant data structure, quadtrees, medial axis transformation

Received 5/84; revised 3/85; accepted 1/86



Authors' Present Addresses: David S. Scott, Department of Computer Science, University of Texas, Austin, TX 78712. S. Sitharama Iyengar, Department of Computer Science, Louisiana State University, Baton-Rouge, LA 70803.