# TID - A TRANSLATION INVARIANT DATA STRUCTURE FOR STORING IMAGES

David S. Scott and S. Sitharama Iyengar*

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-16 June 1984

*Louisiana State University at Baton Rouge.

ABSTRACT

There are a number of techniques of representing pictorial information, among which are borders, arrays, and skeletons. Recent research on quadtrees has produced several interesting results in different areas of image processing. Recently Samet [1] applied the concept of skeleton and medial axis transform to images represented by quadtrees and defined a new data structure called QMAT. All the techniques described in the literature for storing pictorial information based on quadtrees suffer from sensitivity to the placement of the origin.

This paper introduces a new structure for storing images called a TID (for Translation Invariant Data structure). TIDs have the following characteristics:

1) A TID is invariant under translation
2) The image does not have to be a square of size $2^n$ X $2^n$. Any square (or rectangular) image can be converted to a TID.
3) The TID of an image will have (many) fewer black nodes than the corresponding quadtree or QMAT.
4) The space/time costs of computing a TID are (almost) linear in the number of black nodes.

Keywords and phrases: Quadtree, TID, image processing, algorithm.

CR categories 3.63, 8.2

## 1. INTRODUCTION

Efficient methods for region representation are important for use in manipulating pictorial information. There are a number of techniques of representing pictorial information, among which are borders, arrays, and skeletons [1]. The quadtree has recently become an important data structure in image processing. The early history of quadtrees may be traced in papers by Alexandridis, Klinger, Dyer, Hunter, Steiglitz and Tanimotto, Pavlidis, ([2], [3], [4], [5], [6] [7]).

Recent research on quadtrees has produced several interesting results in different areas of image processing. For details, see papers by Dyer, Rosenfeld, Samet ([8], [9], [10]). Much work has been done on the quadtree properties and algorithms for translations and manipulations have been derived by Samet ([11], [12], [13] [14], [17], [18]), Dyer and Schneir ([15], [16]) and others. The question of efficient quadtree storage was addressed by Gargantini, Jones, Iyengar, Grosky, Jain and Samet [19-23].

Recently Samet [1] 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 which are powers of 2. This definition is due to Samet. For more on this refer to [1].

The data structures proposed by Samet, Dyer, Rosenfeld, Jones, Iyengar, Gargantine and others, can be very sensitive to the placement of the origin. In this paper, we demonstrate the usefulness of new structure called a TID (Translation Invariant Data structure) with the following characteristics.

(1) TIDs do not have to start with a square of size $2^n \times 2^n$. Any square (or rectangular) array of pixels can be turned into a TID.

(2) The TID of an image will generally have many fewer black nodes than other corresponding structures.

The main advantage of a TID over other structures (Gargentine, Iyengar, Samet) is that it is translation invariant. Details will be discussed in Section 3 of our paper.

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 defined TIDs and presents a formal algorithm for computing them (which will be presented in pseudo-Pascal). Section 4 discusses storing and searching TIDs efficiently. Section 5 summarizes the advantages and disadvantages of TIDs.

## 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 descendants) 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, and Southeast. In turn, their offspring each represents a subquadrant of the four quadrants and so on until every terminal node represents a region which 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 labelled 'GRAY' and leaves are either 'BLACK' or 'WHITE.'

Various improvements in quadtrees have been suggested including forest of quadtrees [20, 21], hybrid quadtrees [23, 24], linear quadtrees [19], optimal quadtrees for image segments [21]. All of these methods try to optimize quadtrees by removing some or all of the grey and white nodes. All of them maintain the same number of black nodes.

Recently Samet [1] 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) which are not disjoint in general. Black nodes in 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.

## 2.2 Sensitivity of Placement

All the methods of representing images given above suffer from sensitivity to the placement of the origin. Two images which are translations of each other can give rise to very different looking structures. We examine this phenomenon for the above methods using the example of a $2^{n-1}x2^{n-1}$ black square embedded in a $2^n x2^n$ image. In case A the black square is in the upper left corner while in case B it is translated down and right by one pixel. Figure 1 shows cases A and B for n=3.

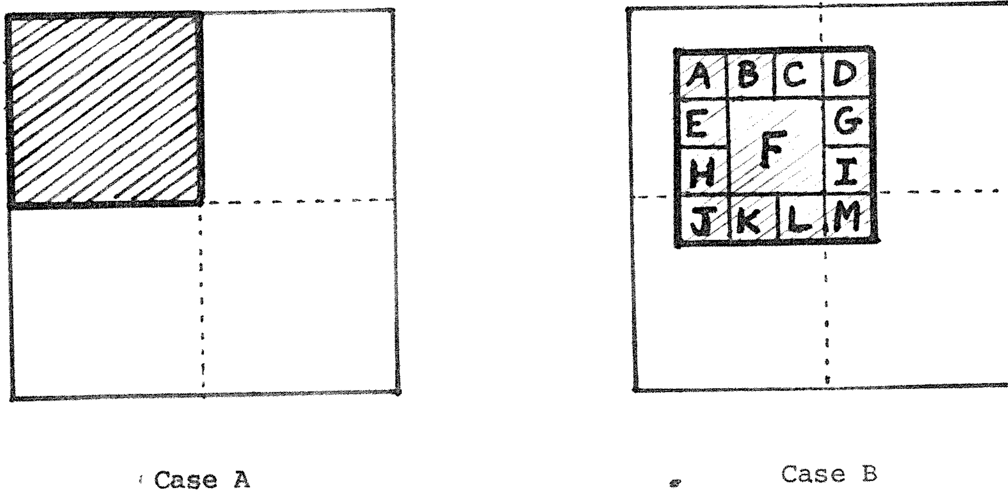



Figure 1. Sample regions

Figure 2 gives the quadtrees for these two patterns and Figure 3 gives the number of each kind of node.

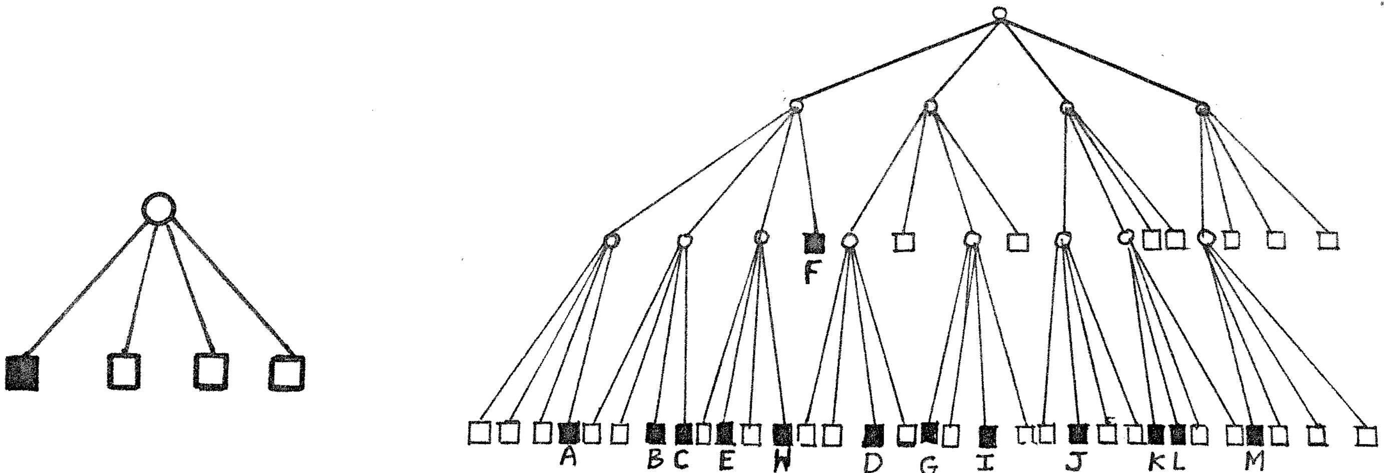



Figure 2. Quadtrees for Case A and Case B of Figure 2.

| | CASE A | CASE B |
|---|---|---|
| Grey Node | 1 | 13 |
| White Node | 3 | 27 |
| Black Node | 1 | 13 |

Figure 3: Number of each kind of node in the quadtrees for n=3.

Column A in figure 3 is constant for all values of n. We will now derive the entries in column B for arbitrary n.

THEOREM 1. The quad tree for placement B has $2^{n+1}-3$ grey nodes, $2^{n+1}+2^{n}+3n-6$ white nodes, and $2^{n+1}+2^{n}-3n-2$ black nodes.

Proof: To prove the formulas it is necessary to derive recurrence relations among certain subtrees.

Let $g_i(n)$, $b_i(n)$, and $w_i(n)$ (for i=1, 2, ..., 5) be the number of grey, black, and white nodes respectively in the quadtrees representing specific $2^n x 2^n$ arrays of pixels. The five arrays of interest are given in Figure 4.

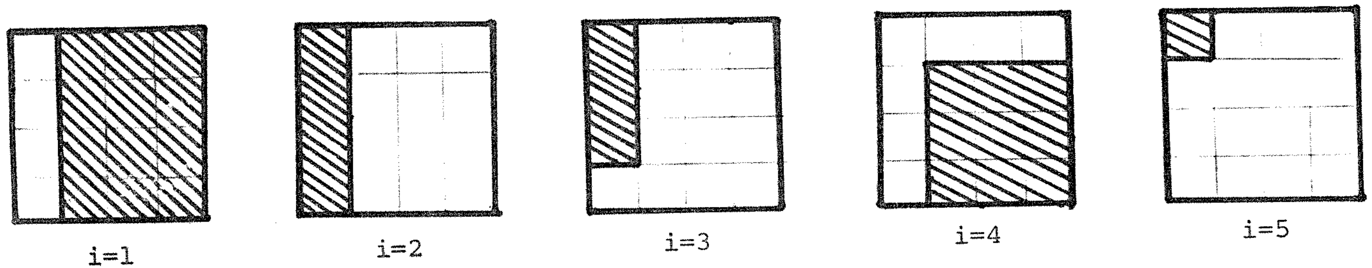



Figure 4.

The following recursion formulas are satisfied

1a) $g_1(n) = 1 + 2g_1(n-1)$

b) $b_1(n) = 2 + 2b_1(n-1)$

c) $w_1(n) = 0 + 2w_1(n-1)$

2a) $g_2(n) = g_1(n)$

b) $b_2(n) = w_1(n)$

c) $w_2(n) = b_1(n)$

3a) $g_3(n) = g_1(n)$

b) $b_3(n) = b_2(n)-1$

c) $w_3(n) = w_2(n)+1$

4a) $g_4(n) = 1 + 2g_1(n-1) + g_4(n-1)$

b) $b_4(n) = 1 + 2b_1(n-1) + b_4(n-1)$

c) $w_4(n) = 2w_1(n-1) + w_4(n-1)$

5a) $g_5(n) = 1 + g_5(n-1)$

b) $b_5(n) = b_5(n-1)$

c) $w_5(n) = 3 + w_5(n-1)$

For example 1a) comes from the fact that subdividing the image yields one grey node, two black squares, and two subimages of type 1.

The boundary conditions are

$g_1(0) = 0$ $g_2(0) = 0$ $g_3(0) = 0$ $g_4(0) = 0$ $g_5(0) = 0$

$b_1(0) = 0$ $b_2(0) = 1$ $b_3(0) = 0$ $b_4(0) = 0$ $b_5(0) = 1$

$w_1(0) = 1$ $w2(0) = 0$ $w_3(0) = 1$ $w_4(0) = 1$ $w_5(0) = 0$

The solutions are

$g_1(n) = 2^n-1$ $g_2(n) = 2^n-1$ $g_3(n) = 2^n-1$

$w_1(n) = 2^n$ $w_2(n) = a^{n+1}-2$ $w_3(n( = 2^{n+1}-1$

$b_1(n) = 2^{n+1}-2$ $b_2(n) = 2^n$ $b_3(n) = 2^n-1$

$g_4(n) = 2^{n+1}-n-2$ $g_5(n) = n$

$w_4(n) = 2^{n+1}-1$ $w_5(n) = 3n$

$b_4(n) = 2^{n+2}-3n-4$ $b_5(n) = 1$

Finally the quadtree for case B can be seen to have one subtree $(2^{n-1} x 2^{n-1})$ of type 4, two of type 3, and one of type 5. Thus, the case B quadtree has

$$1 + g_4(n-1) + 2g_3(n-1) + g_5(n-1) =$$
$$1 + 2^n-(n-1)-2 + 2(2^{n-1}-1) + n-1 = 2^{n+1}-3$$

grey nodes. Similarly it has

$$b_4(n-1) = 2b_3(n-1) + b_5(n-1) = 2^{n+1} + 2^n-3n-2$$

black nodes and

$$w_4(n-1) + 2w_3(n-1) + w_5(n-1) = 2^{n+1} + 2^n + 3n-6$$

white nodes. □

OBSERVATION 1

The quadtree for case B grows exponentially in n while the quadtree for case A has 5 nodes independent of n. How do the various representation schemes apply to case B? Most of the schemes eliminate some or all of the pointers and white nodes but do nothing to black nodes. Thus linear quadtrees, compact quadtrees, hybrid quadtrees, and forests of quadtrees all are inevitably forced to store $2^{n+1} + 2^{n} - 3n - 2$ black nodes plus perhaps some others. The only scheme capable of eliminating black nodes is Samet's QMAT.

OBSERVATION 2

The QMAT for case B 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 5 shows the 172 black blocks for case B 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.

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

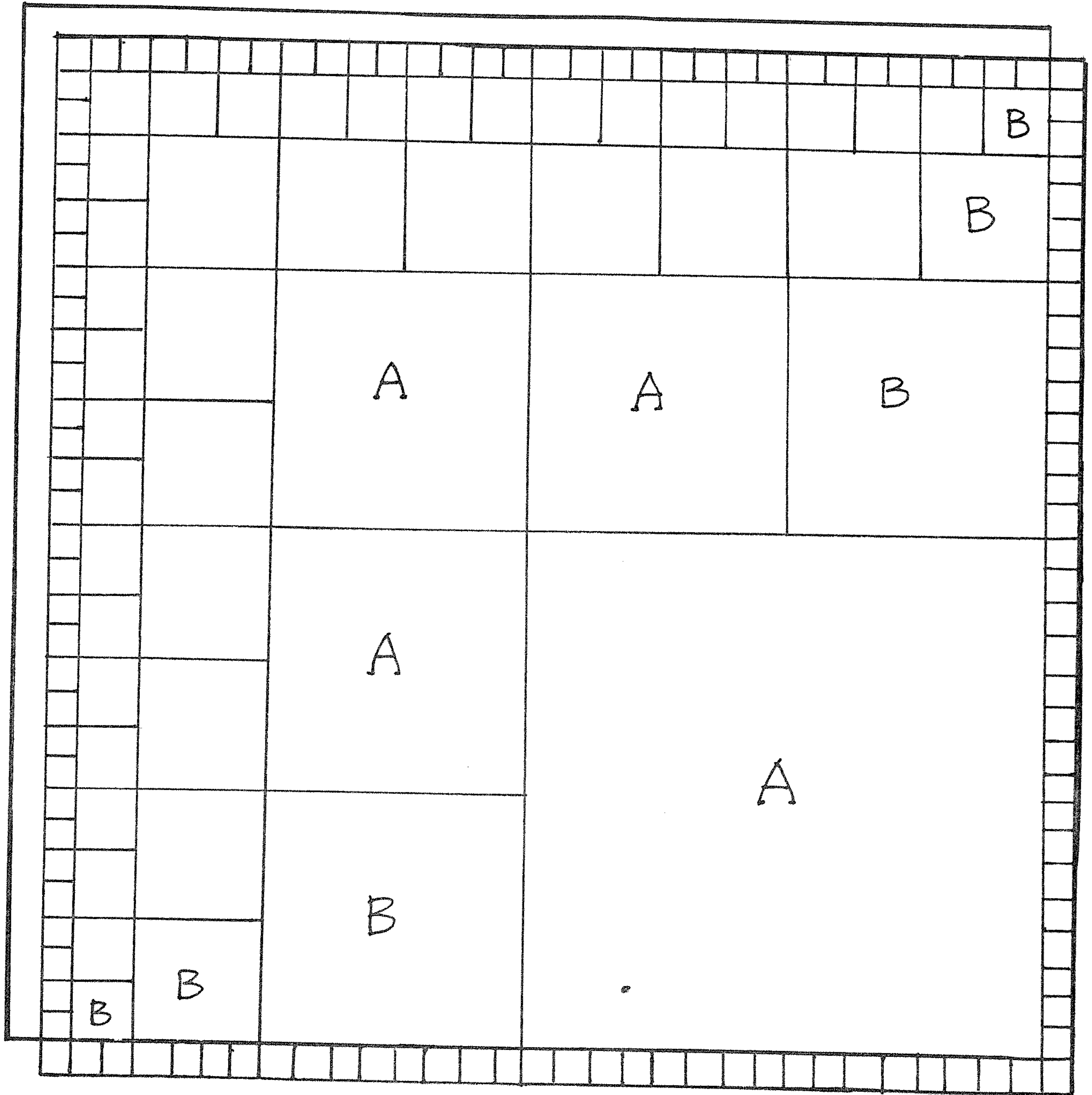
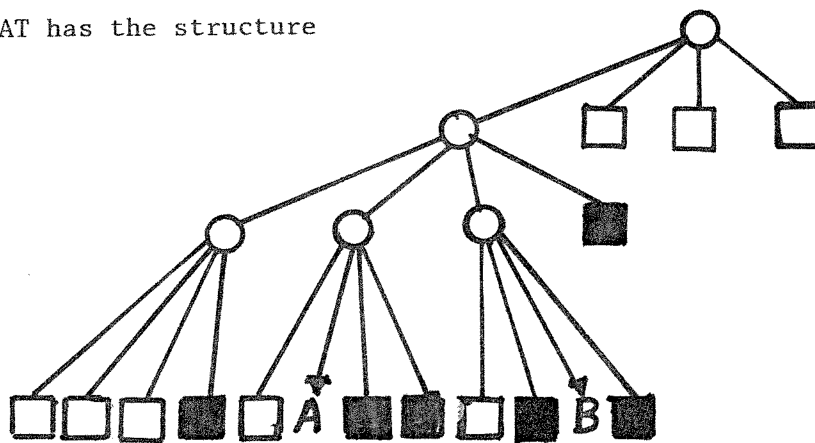



Figure 5: Shows the 172 black blocks For Case B with n=6.

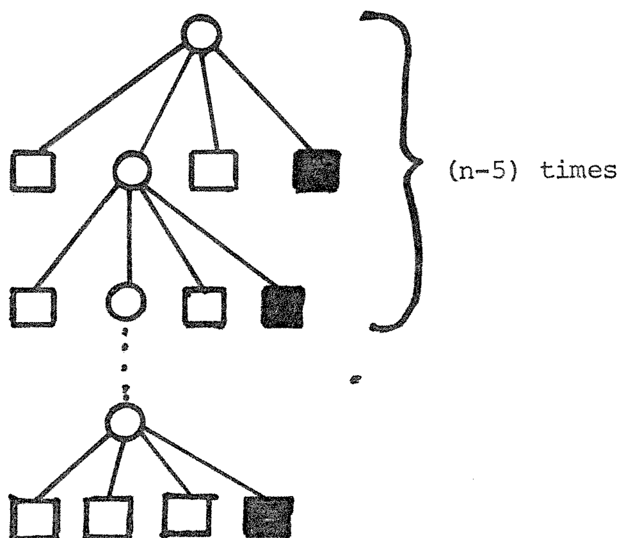
<u>THEOREM 2</u>. For $n \geq 5$ the QMAT for case B has

> $2n-2$ black nodes
>
> $4n-6$ white nodes and
>
> $2n-3$ grey nodes.

<u>Proof</u>. The QMAT has the structure




The subtrees A and B are reflections of each other and A has the form




Thus the number of black nodes is $2x(n-5) + 8 = 2n-2$

The number of white nodes is $4(n-5) + 14 = 4n-6$

The number of grey nodes is $2(n-5) + 7 = 2n-3$.

□

Sensitivity to the placement of the origin is particularly annoying when translating images (for example when several images are combined). As the above example 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 which is translation invariant.

# MAIN RESULT

## 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.

The medial axis transform with respect to the maxnorm ($\infty$-norm) 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 3.5 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). This is not a simple task which may explain why MATs have not been investigated previously in this context.

The essential problem is shown in figure 3.2.1. All the pixels are black except A, B, and C. A is the closest white pixel to

D, A and C are equally distant from E but C is closest to F and B is irrelevant. Any algorithm which sequential processes D, E, and F must be capable of moving smoothly from considering A to considering C while ignoring B.

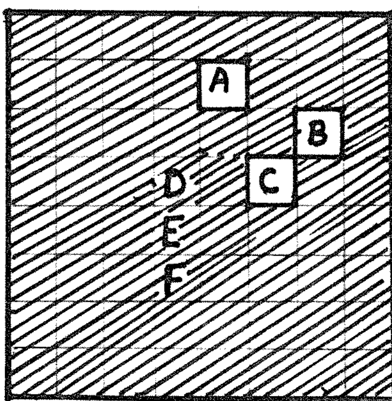



Figure 3.2.1 Sample region to compute the distance from each black pixel to the nearest white pixel (A,B,C).

One approach is to compute the distance from each white pixel to every black pixel and simply maintain the minimum distance at each black pixel. This is easy to code but it is a quadratic algorithm. The algorithm we present here is more complicated but it is also more efficient.

The algorithm to compute the maxnorm distances has two stages. In the first stage the pixels are accessed by row and each black pixel is assigned the distance to the nearest white pixel (or boundary) in its row. The first stage algorithm is simple. Pixels in a row are visited from left to right and the distance from the closest white pixel on the left is recorded. Whenever a white pixel is encountered

after K black pixels, the last K/2 black pixels are revisited to replace their horizontal distances with the smaller distance from the white pixel just encountered.

The algorithm for stage 1 follows.

Each pixel a[i, j] has three fields

Color (black or white)

horiz (horizontal distance to white)

dist (the final distance to white)

---

```
procedure rowpass(a,numrow,numcol)
(* stage 1 - set horizontal distances)
   begin
   for i:=1 to numrow do
        begin
           lastwhite:=0;
           for k:=1 to numcol do
              if a[i,k].color=black
                 then a[i,k].horiz:=k-lastwhite  (* set preliminary value *)
                 else begin
                    if lastwhite <> k-1 then left (i, lastwhite, k,a);
                    a[i,k].horiz:=0;
                    lastwhite:=k
                    end;
             if lastwhite ≠ numcol then left (i, lastwhite, numcol+1, a);
        end;
   end;
```

---

```
procedure left (i,j1,j2,a)
(* back up to set final horiz values *)
begin
   m=j2-((j2-j1) DIV 2);
   for k:=j2-1 downto m do
      a[i,k].horiz:=j2-k;
end;
```

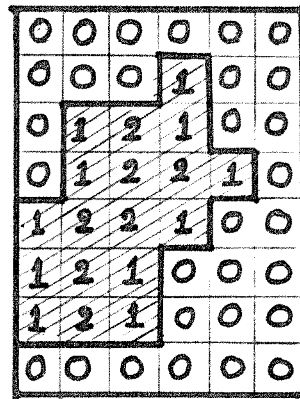



Figure 3.2.2 Result of applying rowpass algorithm to a particular image.

It should be clear that the above algorithm in linear in the number of pixels. In particular, each white pixel is visited once, each black pixel is visited either once or twice, and at most half of the black pixels are visited twice. Figure 3.2.2 shows the result of applying rowpass to a particular image.

The second stage of the algorithm is more complicated. All the information needed to correctly compute the distance function for all the pixels in a column is contained in the column. However it seems to be impossible to do the calculation in time which is linear in the number of pixels in the column. As in stage 1, the stage 2 algorithm will visit each pixel from top to bottom, computing the distance to the nearest white pixel above the current pixel, back-tracking whenever a white pixel is encountered after a black pixel to set the final value. In the horizontal algorithm all distances are computed with respect to a fixed white pixel (lastwhite) until the next

white pixel is encountered. Unfortunately this is not true for the vertical algorithm as shown in figure 3.2.1.

For a fixed column (say the ith), let horiz(j)=horiz(j,i) be the horizontal distance from (j,i) to the nearest white pixel in the jth row (as computed by stage 1). (If the (j,i) pixel is white, then horiz(j,i)=0.) For k>j the distance from the white pixel in the jth row to (k,i) is

$$D(k,j,i)=D(k,j) = \max(\text{horiz}(j),k-j). \qquad (3.2.1)$$

As k increases, D(k,j) is constant (equal to horiz(j,i)) until k-j=horiz(j,i). D(k,j) increases linearly for all larger values of k. The distance recorded at (k,i) should be the minimum value of D(k,j,i) for all $j \leq k$.

Definition: For fixed k (and i) the row j for which the minimum distance occurs is called the active row. (If several rows yield the minimum distance, choose the largest row index.)

Thus for each k it is necessary to find the active row. The following results makes this process easier.

Lemma 1. If $j<\ell$ and $D(k,\ell)\leq D(k,j)$ then for all $m\geq k$, $D(m,\ell)\leq D(m,j)$.

Proof. Induction.

The basis step is part of the hypothesis.

Assume that $D(m,\ell)\leq D(m,j)$, i.e.

$$\max(\text{horiz}(\ell),m-\ell)\leq\max(\text{horiz}(j),m-j) \qquad (3.2.2)$$

The only way that $D(m+1,\ell)>D(m+1,j)$ can occur is if the following 3 equations hold

(1) $D(m+1,\ell)=D(m,\ell)+1$

(2) $D(m+1,j)=D(m,j)$

(3) $D(m,\ell) = D(m,j)$

From (1) we get $\text{horiz}(\ell)\leq m-\ell$

From (2) we get $\text{horiz}(j)>m-j$

From (3) we get $\text{horiz}(j)=m-\ell$

Combining the last 2 inequalities yields $j>\ell$ which is a contradiction.. □

<u>Corollary 1</u>. If $A(k)$ is the active row index of step k then $A(k)$ is an increasing function of k.

<u>Corollary 2</u>. If row j is not yet active at step k and $\text{horiz}(k)=D(k,k)\leq D(k,j)$ then row j will never be active.

These results would seem to lead to a linear algorithm. Simply move down the column from one active row to the next. Unfortunately there is no way to recognize the next active row without examining the column down to k. Instead we will keep a list of potentially active rows. The algorithm has three parts:

1. Add the pair (k,horiz(k)) to the list of potentially active rows.
2. Set the distance for the (k,i)th pixel to $D(k,j)$ where j is the top element of the list.

3. If m is the second element in the list (if it exists) and $D(k,m) = D(k,j)$ then delete j from the list.

The Add process requires more explanation. The list is to contain all the potentially active rows in order with the active row at the top. It is not sufficient just to append row k to the bottom of the list. This would imply that each row in turn becomes active which is contradicted by figure 3.2.1. Rather it may be necessary to delete some rows from the bottom of the list before appending row k. Corollary 2 gives the appropriate criterion for deleting a row from the list. In particular, row j is deleted from the list at step k if

$$D(k,j) \geq D(k,k) = \text{horiz}(k).$$

This insures that rows which can never become active are deleted from the list.

Thus each row is added to the list and either it survives to become an active row, it is deleted from the list when some later row is added, or it may possibly still be in the list when the bottom of the column is encountered.

To finish showing that the algorithm is correct, it is necessary to show that each row which reaches the top of the list is actually the active row. The following lemmas are sufficient.

Lemma 2. If $j_1<j_2<...<j_m$ are the rows in the list at any time, then

$$horiz(j_1)<horiz(j_2)<...<horiz(j_m).$$

Proof. Suppose not. Let $\ell$ and p be two rows on the list with $\ell<p$ but $horiz(\ell)\geq hdist(p)$. Then at step p

$$D(p,\ell)\geq D(p,p)$$

which means that row $\ell$ would have been deleted when row p was added to the list. □

Lemma 3. If $j_1<j_2<...<j_m$ are the rows in the list at the beginning of step k then

$$D(k,j_1)\leq D(k,j_2)<D(k,j_3)<...<D(k,j_m).$$

The first inequality follows from part 1 of each step which deleted j, from the list at any step $\ell$ for which $D(\ell,j_1)=D(\ell,j_2)$ and since distances change by at most from one step to the next it is impossible for $D(k,j_1)>D(k,j_2)$ without having $D(\ell,j_1)=D(\ell,j_2)$ for some previous $\ell$.

Suppose $D(k,j_1)\leq D(k,\ell)\geq D(k,p)$ for $\ell<p$ both in the list. Examining the second inequality we find

$$\max(horiz(\ell),k-\ell)\geq \max(hdist(p),k-p)$$

By assumption $\ell<p$ and by Lemma 2 $horiz(\ell)<horiz(p)$ and so we must have

$$horiz(\ell)<k-1 \qquad (*)$$

$$horiz(p)>k-p$$

$$k-\ell\geq horiz(p)$$

From the first inequality (*) we have

$$\max(horiz(j_1),k-j_1)\leq \max(horiz(\ell),k-\ell) \qquad (**)$$

By lemma 2 $horiz(j_1)<horiz(\ell)$ and by (*) $horiz(\ell)<k-\ell$. Substituting these in (**) yields

$$k-j_1\leq k-\ell$$

which contradicts $j_1<\ell$. □

Thus the distances are strictly increasing except perhaps for the first one. Thus the active row is always at the top of the list except when equality occurs compared to the second element. This is precisely when part 3 of the algorithm will delete the top element and restore the invariant that the active row is at the top of the list.

THEOREM 3. For each black pixel the above algorithm correctly computes the distance to the nearest white pixel which is in the same or earlier row.

Proof: Discussion above.

It is clear that after a downward pass through a column, it is necessary to have an upward pass to finish computing the correct distance. The only difference in the upward pass is that the extra information available from the downward pass (that is the distance to the closest white pixel above the current one) allows some potential savings.

Lemma 4. If at step k of the upward pass horiz(k)> dist(k) then the distance computed in the downward pass, then row k need not be added to the list.

Proof: There must be some $j<k$ with $D(k,j)<horiz(k)$ in which case $D(m,j)<D(m,k)$ for all $m<k$ and so row k will not influence any of these distance values. □

The formal algorithm for the vertical pass follows. The additional data structure needed is LIST. LIST is a record with three fields TOP, BOTTOM, and ROWS. TOP and BOTTOM are integer pointers and ROWS [1..NUMROW,1..2] is an array of integers. TOP points to the active row, BOTTOM points to the bottom of the list, and the two elements in ROWS are of the form k, horiz(k). LIST starts with

BOTTOM=1, TOP=1 and ROWS(1,1)=0 and ROWS(1,2)=0. This zero row represents the virtual white pixel at the boundary.

```
procedure columnpass(a,numrow,numcol,list)
(* Stage 2 vertical pass - sets distances *)
begin
for i:=1 to numcol do
   begin
      lastwhite:=0;
      init(list,0);
      for k:=1 to numrow do
         if a[k,i].color=black
            then begin
               add (k,a[k,i].horiz,list);              (* part 1 *)
               a[k,i].dist:=distk(k,list.top,list)'    (* part 2 *)
               checklist(list)                         (* part 3 *)
               end
            else begin
               if lastwhite ≠ k-1
                  then up(i,lastwhite,k,list,a);
               init(list,k);
            end;
         if lastwhite ≠ numrow then up(i,lastwhite,numrow +1, list,a);
      end;
end;
```

the subsidiary modules follow:

```
procedure init (list,k)
(* initialize list with a white pixel in row k *)

begin
      list.top:=1
      list.bottom:=1;
      list.rows[1,1]:=k;
      list.rows[1,2]:=0;
end;
```

The next module of the algorithm computes the infinity norm distance from k to row $\ell$ of list.

```
function distk(k,ℓ,list)
(* compute infinity norm distance from k to row ℓ of list *)
distk:=max(abs(k-list.rows[ℓ,1]), list.rows [ℓ,2]);
```

---

```
procedure add(k,horizk,list)
(* locate place to add new row *)
begin
   j:=list.top;
   done:=false;
   while not done do                (* linear search for insertion point *)
   if j > list.bottom
      then begin
         assign (k,horizk,j,list);
         done:=true
      end
   else
      if hdistk ≤ distk(k,j,list)
         then begin
            assign (k,horizk,j,list);
            done:=true
         end
      else j:=j+1;
end;
```

```
procedure assign (k,hdistk, j,list)
(* put row k in list *)
begin
      list.bottom:=j;
      list.rows[j,1]:=k;
      list.rows[j,2]:=hdistk
end;
```

```
procedure checklist (k,list)
(* delete top of list if appropriate *)
begin
   if list.top ≠ list.bottom
   then if distk(k,list.top,list)=distk(k,list.top + 1,list)
      then list.top:=list.top + 1;
end;
```

```
procedure up(i,j1,j2,lista)
(* make an upward pass from j2-1 back to j1+1 *)
begin
   init(list,j2);
   for k:=j2-1 downto j2+1 do
      begin
         if a[k,i].dist≥a[k,i].horiz then add (k,a[k,i].horiz,list);  (*part 1*)
         a[k,i].dist=min(a[k,i].dist,distk[k,list.top,list]);         (*part 2*)
         checklist(k,list);                                            (*part 3*)
      end;
end;
```

The stage 2 algorithm is almost linear in the number of pixels. Each white pixel is visited once and each black pixel is visited twice. The amount of work at each step is constant except in the procedure ADD. ADD must search a list to determine where to place row k. The length of the list is not bounded by a constant and may have O(numrow) entries in it. The procedure ADD given above uses a sequential search, for simplicity, which yields O(numrow) bound on time. Thus the total time for the algorithm as given is bounded by $O(numcol*numrow^2)$. However the list is ordered and stored in an array and so the given ADD could be replaced by a binary search whch has a time bound of O(log(numrow)). Thus the time bound for the entire algorithm could be reduced to O(numcol*numrow*low(numrow)). Finally if numrow<numcol the stages of the algorithm could be reversed. So we have:

THEOREM 4. The time required by the distance algorithm is $O(numrow*numcol*log(min(numrow,numcol)))$.

Proof. Discussion above.

Even the above bound is pessimistic. We now show that if the horiz values are independent then the expected length of the list is only log(numrow) even ignoring the fact that rows are eventually deleted from the top of the list.

Theorem. Assuming that all permutations of the horiz values are equally likely, then the expected number of elements in list is $\theta(log(numrow))$ after numrow adds.

Proof. Start from the bottom of the column and work up. The bottom row is in list with probability 1. The next to bottom element has probability of ½ of being in the list, and so on. Thus the expect length of the list is

$$\sum_{i=1}^{numrow} \frac{1}{i}$$

which is well known to be $log(numrow)+\theta(1)$.

Cor. The expected running time of the algorithm to compute maxnorm distances is

$$\theta(numrow*numcol*log(log(min(numrow,numcol))))$$

## 3.3 Locating Maximal Squares

Let D(i,j) be the distance from (i,j) to the nearest white pixel as computed by the algorithm in section 3.2. 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 2x2 square of pixels which 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 5 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.

(b) (i,j) is not part of a constant 2-square.

Proof. (—>) Let Q be a maximal square. Suppose Q has even order and let T be the 2-square 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 constant. This completes the 1-part 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 2a. 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 2b. This completes (—>)

(<—) Let Q be the square centered on the constant 2-square T and suppose Q is strictly contained in some larger black square S. Let (i,j) εT and (k,m)εT be such that (i,j) is closer to the center of S then (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 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 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 order square centered on $(k,m)$ strictly contains Q. This is the previous case which contradicts 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 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 2b. Hence Q is maximal. □

The characterization of maximal squares given in Theorem 5 is local. This leads immediately to a linear algorithm for locating maximal squares. For each black pixel, simply examine all adjacent pixels to determine whether 1 or 2 holds. The algorithm can be improved somewhat by reducing the constant. For reasons which will be made clear in the next section, the algorithm presented below only locates which pixels are local maxima of D. The algorithm processes the rows from top to bottom and left to right in each row. The current distance value is always compared to its W and NE neighbors.

Depending on the comparison with the W neighbor it may or may not be compared to its NW and N neighbor. The algorithm requires special processing of the first row, first column, and last column to avoid illegal array subscripts.

```
procedure maxima(a,numrow,numcol)
(* mark local maxima by horiz=1 and horiz=0 otherwise *)
begin
(* special first row *)
   for k:=1 to numcol do a[1,k].horiz:=a[1,k].dist;
(* other rows *)
   for i:=2 to numrow do
      begin
(* special first element of row *)
         a[i,1].horiz:=a[i,1].dist;
         if a[i-1,2].dist=2 then a[i,1].hdist:=0;
(* generic row element *)
         for k=2 to numcol-1 do test (i,k);
(* special last element *)
         a[i,numcol].horiz:=a[i,numcol].dist;
         if a[i,numcol-1].dist=2 then a[i,numcol].horiz:=0;
end;
```

```
procedure test (i,k)
begin
   a[i,k].horiz:=1;
   if a[i,k].dist = 0
      then a[i,k].horiz:=0
      else begin
(* test NE neighbor *)
      if a[i,k].dist<a[i-1,k+1].dist then a[i,k].horiz:=0;
      if a[i,k].dist>a[i-1,k+1].dist then a[i-1,k+1].horiz:=0;
(* test W neighbor and (NW and N if necessary) *)
      if a[i,k].dist<a[i,k-1].dist
         then a[i,k].horiz:=0
         else
            if a[i,k].dist=a[i,k-1].dist
               then begin
                  if a[i,k].dist<a[i-1,k-1].dist then a[i,k].horiz:=0; (* NW *)
                  if a[i,k].dist<a[i-1,k].dist then a[i,k].horiz:=0;   (* N *)
               end
         else begin      (* a[i,k].dist>a[i,k-1].dist *)
            a[i,k-1].horiz:=0;
            if a[i,k].dist>a[i-1,k-1].dist then a[i-1,k-1].horiz:=0;
            if a[i,k].dist>a[i-1,k].dist then a[i-1,k].horiz:=0;
      end;
   end;
end;
```

## 3.4 Eliminating Redundant Maximal Squares

Not all maximal squares may be needed to cover an image. Figure 3.4.1 displays a black rectangle which is the union of two squares. However the image contains six maximal squares (each centered on a 3). It is clearly desirable to eliminate these redundant squares. The squares at the ends are needed and all of the others are redundant. 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 3.4.2).

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 3.4.1 Displays a black rectangle which is the union of two squares.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 3.4.2 Expansion of the rectangle described in Figure 3.4.1

A maximal square is <u>redundant</u> if it is covered by the rest of the maximal squares.

For simplicity we will first discuss determining whether an odd order maximal square is redundant. A maximal square Q is redundant if it is covered by two or more maximal squares. The most common situation is when Q is covered by two squares (from opposite sides). The two covering squares may be the same size as Q (figure 3.4.1), larger than Q (figure 3.4.3), or mixed (figure 3.4.4). It is impossible to cover Q with only two squares if one of them is smaller than Q.

| 1 | 1 | 1 | 1 | 1 | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 1 | | | | |
| 1 | 2 | 3 | 2 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 1 |
| | | | | 1 | 2 | 2 | 2 | 1 |
| | | | | 1 | 1 | 1 | 1 | 1 |

Figure 3.4.3

| 1 | 1 | 1 | 1 | 1 | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 1 | | |
| 1 | 2 | 3 | 2 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 3.4.4

Let Q be the redundant square and let (i,j) be its center. In all three figures we see that D(i,j-1)=D(i,j)=D(i,j+1). We now prove that this (or the vertical analog) always occurs if Q is covered by two other squares.

Lemma 1. 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.

Proof. (<——) By symmetry we may assume a) holds.
Let S. by the square centered on (i,j-1) and let T be the square centered on (i,j+1). S covers all of Q except the right most column. T covers all of Q except the left most column. Since D(i,j)>1, Q_SUT. Neither S nor T are contained in Q and hence each must be contained in some other maximal square. Hence Q is covered by two maximal squares.

(—>) If D(i,j)=1 then Q intersects no other maximal square and so D(i,j)=1 is impossible. Let Q_SUT where Q, S, and T are distinct maximal squares. Since Q_S, there must be some edge of Q which S does not cover at all, say the right edge. T must cover the entire right edge and hence D(i,j+1)≥D(i,j). Since Q is maximal D(k,j+1)>D(i,j) is impossible and so D(i,j+1)=D(i,j). Since Q_T, T must not cover any of the left edge of Q. Hence S covers the entire left edge of Q and so D(i,j-1)=D(i,j). If S doesn't cover the bottom (or top) edge of Q then D(i-1,j)=D(i,j)=D(i+1,j) instead. □

Lemma 1 gives a simple local characterization of redundant squares which are covered by 2 other squares. Unfortunately, it is much harder to characterize maximal squares which take 3 or more squares to cover them. Figures 3.4.5, 3.4.6, and 3.4.7 give three examples of such squares.

| 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 1 | 0 | 0 |
| 1 | 1 | 2 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |

Figure 3.4.5: Examine the upper right corner of the red square for maximal square.
( The big red square is covered by the 3 small red squares)

| 0 | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 2 | 1 | 0 |
| 1 | 2 | 2 | 2 | 2 | 1 | 0 |
| 1 | 2 | 3 | 3 | 2 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 3.4.6: Examine the entire right edge of the red square for maximal squares.
( The big red square is covered by the 3 small red squares)

| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 0 |
| 1 | 2 | 2 | 2 | 2 | [illegible] | 2 | 1 | 0 |
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 | 1 |
| 1 | 2 | 2 | 3 | 4 | 3 | 2 | 2 | 1 |
| 1 | 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 2 | 2 | 2 | 2 | 1 |
| 0 | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Figure 3.4.7: Examine the whole square for maximal squares.
( The big red square is covered by the 4 small red squares)

As can be seen from the figures there is no local way of determining whether a maximal square is covered by 3 or more other maximal squares. In figure 3.4.5 it is necessary to examine the upper right corner of the red square. In figure 3.4.6 it is necessary to examine the entire right edge of the red square and in 3.4.7 it is necessary to examine the whole square! These cases are much rarer than when only two squares are needed. Furthermore if we modify figure 3.4.5 to figure 3.4.8 we see that the red square is still

| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 1 | 0 | 0 |
| 1 | 1 | 1 | 2 | 1 | 0 | 0 |
| 0 | 0 | 1 | 2 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |

figure 3.4.8: Modification of figure 3.4.5 to show that the red square is still redundant.

redundant but that it would be better to keep the red square and eliminate the two squares centered on the circled 2's. In general there is a complex set of dependencies which are expensive to compute. Once having computed them it is then a difficult task to determine an optimal set of squares to delete.

For these reasons only redundant squares which can be covered by 2 other squares will be considered for deletion. By lemma 1 such

squares are always signalled 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 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 larger square contains them.

Figure 3.4.9 shows an example with k=2 and n=8. Note that both endpoints are not local maxima. 2k-2=2 so two squares are

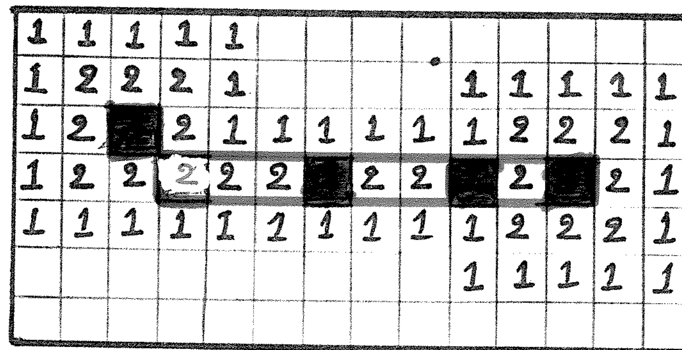



Figure 3.4.9

deleted for each one kept. If the algorithm starts at the left end the circles centers are kept. The endpoints are covered by the circled 3's. Obviously which squares are kept may depend on whether the algorithm starts at the left or right end but the number of squares deleted is the same.

Assuming that the algorithm always goes left to right (and top to bottom) then 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 3.4.10, 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.

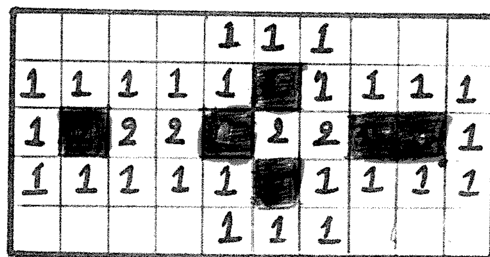


figure 3.4.10 Example Region

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. The 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 yes then do not delete the square and start the delete count over (i.e. delete te next 2k-2 squares). The vertical pass is unaffected.

There remains the question of even order maximal squares. As shown in Theorem 5, an even order maximal square is characterized as centered on a constant 2 square. As for odd order maximal squares, there are a variety of ways in which an even order maximal square might be redundant. We will only consider two of them.

THEOREM 6. 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 $T \leq R \quad S$.

Proof. If a pixel P is not a local maximum of D then the square centered 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 2x8 rectangle of constant D values with T being the central square. Then Q is covered by the squares centered on R and S and Q is redundant. □

Theorems 5 and 6 together indicate that we need only concern ourselves with pixels which are local maxima of D. Redundant squares in sequences of equal squares (or equal 2-squares) can be handled in the same way. The code given in section 3.3 marks local maxima with horz=1. The follow code scans the pixels looking at local maxima to see if 1) they can be deleted as part of a constant sequence and 2) whether they can be expanded to constant 2-squares. Possible horizontal sequences are checked first for possible deletions followed by vertical sequences. Essential constant 2-squares are marked by setting the horz values to

<table><tr><td>3</td><td>2</td></tr><tr><td>2</td><td>2</td></tr></table>

. If two essential constant 2-squares overlap then they will end up marked as

<table><tr><td>3</td><td>3</td><td>2</td></tr><tr><td>2</td><td>2</td><td>2</td></tr></table>

or

<table><tr><td>3</td><td>2</td></tr><tr><td>3</td><td>2</td></tr><tr><td>2</td><td>2</td></tr></table>

.

Thus the value 3 is used as a marker for the NW corners of essential constant 2-squares.

```
procedure deletesquares (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 *)
(* 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+1].dist=d
            then begin
               count:=count+1;
               a[i,k].horiz:=0
            end;
```

```
procedure deletecols (a, numrow, numcol)
(* delete unneeded local maxima by scanning columns *)
begin
   for k:=2 to numcol-1
        begin
            d:=0;
            for i:=1 to numrow-1 do
                if a[i,k].horiz≠1
                    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);
                           if a[i,k].horiz≠3 then twosquare (i,k-1,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].horz:=0
                      end;
    end;
end;
```

```
procedure twosquare (i,j)
(* check if 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;
```

3.4 Storing and Searching a TID

Each maximal square in a TID is characterized by three number (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, and 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 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 orderings

a) (i,j,s)

b) (i,s,j)

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 subdivision 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 b and c. For most images c is superior since there will be many small squares around and so the subsets of size 1, 2, and 3 should 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, i.e. 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 b we can stop the search as soon as $k<i$. There remains the question of whether plan a or plan b allows more skipping of squares. For a 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 following assumptions:

1) Every value of j is equally likely.
2) All possible values of s (i.e. 1 to numrow-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 $(\ell,j,s)$ with $\ell=i$ and $j>m$. By assumption 1 this will be about half the squares with $\ell=i$ for a random (k,m). Plan b will skip all squares $(\ell,j,s)$ with $\ell=i$ and

$\ell+s<k$. By assumption 2 this will be about half the squares with $\ell=i$. Thus by analysis the two plans are about the same. However both biases in the assumptions 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. Pros and Cons of TIDs

(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 being combined into one composite.

(2) A second advantage 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 imbed it in a square of 2-power order.

For example, if a black square of order $(2^k+1)x(2^k+1)$ needs to be stored, the TID would just be the black square. To store it as a quadtree requires that it be imbedded in a $2^{k+1}x2^{k+1}$ square. The best imbedding would require $2^{k+1}+2$ black leaves.

(3) The third 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 in [CACM March 84 p. 248-9] Markku Tamminen quotes 5,198 black leaves for the quadtree encoding the circle inscribed in a $2^{10}x2^{10}$ square. The corresponding TID has 601 black squares, an 88% reduction.

The computation cost of TIDs is little higher than quadtrees. Although the cost is almost linear in the number of pixels the constant is not as small as it is for quadtrees. A quadtree can be computed by examining each pixel only once. Procedure distance examines each white pixel twice and each black pixel at most 4 times. Procedure maxima examines about 3 pixels for each black pixel and

procedures maxsquare examines about 8 pixels for each local maximum of the distance function D. Maxima and Maxsquare could be overlapped to lower the number of accesses.

In principle the TID for a $2^n x 2^n$ 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.)

We analyze the union of two regions arising (or suitably projected) from different screens and superposition of images using TID is described in our other paper [25]. The reader is referred to our paper for the basic idea behind manipulation algorithms on TID and its relation to other representation.

Representation of three-dimensional digital images using TID structure is presently under investigation and the results will be reported shortly in a different paper. Detailed testing of this method is presently underway and the software 'TIDSOFT' will be available for users by this summer (84).

# References


[1] H. Samet, "A Quadtree Medial Axis Transform," Communications of the ACM, Volume 25, No. 9, pp. 680-693, September 1983.

[2] N. Alexandridis and A. Klinger, "Picture decomposition, tree data-structures and identifying directional symmetries as node combinations," Comput. Graphics Image Processing, no. 8, pp. 43-47, 1978.

[3] A. Klinger and C. R. Dyer, "Experiments in picture representation using regular decomposition," Computer Graphics and Image Processing, vol. 5, pp. 68-105, 1976.

[4] G. M. Hunter, "Efficient computation and data structures for graphics," Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, Princeton University, Princeton, NJ, 1978.

[5] G. M. Hunter and K. Steiglitz, "Operations on images using quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 1, pp. 145-153, 1979.

[6] G. M. Hunter and K. Steiglitz, "Linear transformation of pictures represented by quadtrees," Computer Grapics and Image Processing, vol. 10, pp. 289-296, 1979.

[7] S. Tanimoto and T. Pavlidis, "A hierarchical data structure for picture processing," Computer Graphics and Image Processing, vol. 4, pp. 104-119, 1975.

[8] C. R. Dyer, A. Rosenfeld, and H. Samet, "Region representation: boundary codes from quadtrees," Communications of the ACM, vol. 23, pp. 171-179, March 1980.

[9] H. Samet, "Region representation: quadtrees from binary arrays," Computer Graphics and Image Processing, vol. 18, pp. 88-93, 1980.

[10] H. Samet, "Region representation: quadtrees from boundary codes," Communications of the ACM, vol. 23, pp. 163-170, March 1980.

[11] H. Samet, "An algorithm for converting rasters to quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 3, pp. 93-95, 1981.

[12] H. Samet, "Algorithms for the conversion of quadtrees to rasters," to appear in Computer Graphics and Image Processing, 1983 (also University of Maryland Computer Science TR-979).

[13] H. Samet, "Connected component labeling using quadtrees," Journal of the ACM, vol. 28, pp. 487-501, July 1981.

[14] H. Samet, "Computing perimeters of images represented by quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 3, pp. 683-687, 1981.

[15] C. R. Dyer, "Computing the Euler number of an image from its quadtree," Computer Graphics and Image Processing, vol. 13, pp. 279-276, 1980.

[16] M. Shneier, "Path-length distances for quadtrees," Information Sciences, vol. 23, pp. 49-67, 1981.

[17] H. Samet, "Distance transform for images represented by quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 4, pp. 298-303, 1982.

[18] H. Samet, "Neighbor finding techniques for images represented by quadtrees," Computer Graphics and Image Processing, vol. 18, pp. 37-57, 1982.

[19] I. Gargantini, "An effective way to represent quadtrees," Communications of the ACM, vol. 25, pp. 905-910, December 1982.

[20] L. Jones and S. S. Iyengar, "Representation of regions as a forest of quadtrees," Proceedings of the IEEE Conference on Pattern Recognition and Image Processing, Dallas, 1981, 57-59.

[20a] L. P. Jones and S. Iyengar, "Space and time efficient virtual quadtrees," IEEE-PAMI, vol. 6, no. 2, March 1984, pp. 244-247.

[21] W. I. Grosky and R. Jain, "Optimal quadtrees for image segments," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.5, pp. 77-83, 1983.

[21a] D. J. Abel and J. L. Smith, "A data structure and algorithm based on a linear key for a rectangle retrieval problem," to appear in Computer Graphics and Image Processing, 1983.

[22] H. Samet, "Data structures for quadtree approximation and compression," Computer Science TR-1209, University of Maryland, College Park, MD, August 1982.

[23] V. Raman, S. Iyengar and Kundu, "An optimized quadtree structure for pictorial data representation using top and bottom compaction techniques," Proceedings of IEEE-SMC Conference, Bombay, Dec. 1983.

[24] S. Iyengar and V. Raman, "Properties of the Hybrid Quadtrees," to appear in the Proceedings of the 7th International Conference on Pattern Recognition (July 1984).

[25] David Scott and S. Iyengar, "A New Data Structure for Efficient Storing of Images," submitted to Pattern Recognition Letters (June 1984).

[26] Markku Tamminen, "Comment on Quad- and Octtrees, Vol. 27, No. 3, pp. 248-249, March 1984. CACM