RECTANGULAR CODING FOR BINARY IMAGES

Y. C. Kim
J. K. Aggarwal

TR-84-1-23 February 1984

Laboratory for Image and Signal Analysis
Department of Electrical Engineering
The University of Texas
Austin, Texas 78712

This research was supported in part by a grant from IBM Corporation, and in part by the Air Force Office of Scientific Research under Contract F49620-83-K-0013.

ABSTRACT

An algorithm is presented for constructing a rectangular code from the array representation of a binary image. The rectangular code represents the object regions of an image. It is a compact representation and enables efficient storage and transmission of binary images. In addition, for simple operations on the image such as scaling, translation and rotation by multiples of 90 degrees, equivalent operations in the corresponding rectangular code can be determined. The computation of measures such as the area or centroid of an object is easily performed. The execution time of the algorithm is proportional to the number of pixels in the image. The extension of this algorithm to gray level images is straightforward.

1. INTRODUCTION

In general, digitized images are typically represented as 2-D arrays of pixels. However, in operating with 2-D arrays, the storage and processing requirements grow as n^2 for (n x n) images. In addition, 2-D arrays are not always amenable to, or efficient for, the intended operations. By selecting an appropriate data structure for a problem, the operations to be performed may be greatly simplified in terms of the time and storage requirements. These considerations become especially important in a processing environment where it may be necessary to perform several different classes of operations. Use of a single data structure in this environment may result in inefficient use of time and space and thus unacceptable processing or storage overhead. Therefore a considerable interest has been shown in developing data structures for representing binary images. These structures should possess the characteristics of efficient processing and comparatively low storage requirements (over that of the 2-D array) for the intended application. In developing such data structures, two broad approaches may be distinguished --- the first makes use of information at the boundaries of object regions and the second makes use of the area covered by the object.

Most of the earlier work was concentrated on representing binary images with 'boundary information'. Boundary coding and its properties have been studied by Freeman [1]. Montanari [2] reduced the difficulties encountered in extracting a minimal length polygonal contour from a digitized image by first smoothing the contour chains. Following this the polygonal approximations are extracted. Zahn [3] used a similar approach, e.g., smoothed contour lines to describe 2-D

binary patterns. In his method, each of the contour lines is described by its direction and length. Merrill [4] proposed 'TCB' representation scheme (tightly closed boundary). A 'TCB' consists of sets of ordered X-coordinates of boundary points which have the same Y-coordinate. This data structure is efficient for searching areas of the image. Danielsson [5] improved Morrin's [6] and Cederberg's [7] chain coding algorithms by introducing the use of adjacency maps and nesting trees.

More recently, hierachical data structures have been proposed to represent binary images, e.g., the image pyramid [8][9], quadtree [10][11][12][13][14] etc. The second approach described above exploits the fact that a binary image may be represented by descriptions of either of the uniform regions (1's or 0's). In order to describe a uniform region, a certain amount of information has to be encoded in the descriptions. One method would be to introduce a set of elementary shapes which may be used to represent a uniform region. Circles, triangles and rectangles have been proposed as such elementary shapes [15]. A circle requires the least number of parameters (the X and Y coordinates of the center and the radius) to represent a uniform region. However an image array is comprised of discrete samples measured on a rectangular grid. This makes it difficult to represent an object precisely with a set of circles. Rectangles require 4 parameters (the X and Y coordinates of one vertex, width and height) to describe them and represent a natural way to describe uniform regions on a rectangular grid.

In describing regions using rectangles, it is of interest to

minimize the number of rectangles required to describe a given region. A number of research efforts have focused on this problem.

Aoki [15] developed an algorithm to compress image data using rectangular region coding of a binary image. For the rectangular region coding, he used an image pyramid [8][9] of the binary image. The algorithm consists of two main steps.

1. Find and encode the largest possible rectangle in a uniform region. This is repeated for any remaining unencoded area of the image. The choice of rectangles is determined based on a criterion of efficiency [15].

2. Describe the pixels which are not encoded by the rectangles as a set of explicit point-by-point data.

It is pointed out in [15] that repeatedly determining the largest possible rectangle for the remaining unencoded area of a uniform region does not minimize the number of rectangles required to describe the region. A disadvantage of this algorithm is that an image pyramid should be initially constructed. This in itself may require a significant amount of processing.

Ferrari et. al. [16] developed an algorithm to find a minimal rectangular partition of a digitized blob. His formula which finds the minimum number of rectangles is N - L + 1, where N is the total number of concave vertices on the boundary of a digitized blob, and L is the maximum number of nonintersecting chords that can be drawn between cogrid concave vertices [16]. However this algorithm is not applicable to regions with holes. For example, for the object region in Fig 1-1, N is 10 and L is 3. The minimum number of required rectangles is 8 as given by the above formula. However the object region can be partitioned into 7 rectangles.

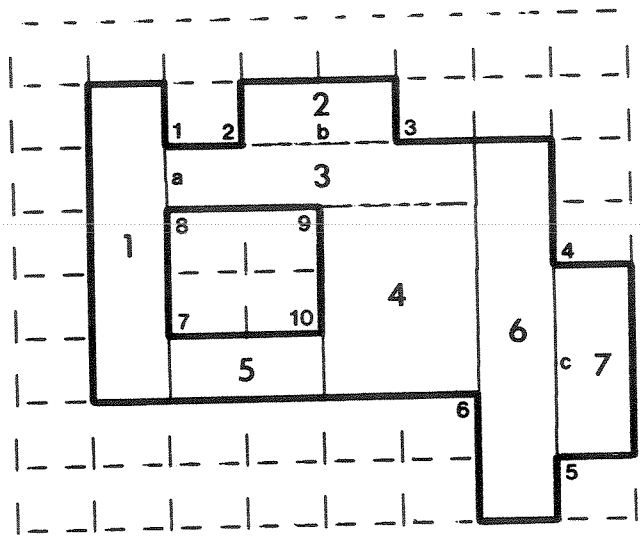


Figure 1-1: An example for which the algorithm in [16] is not applicable.

In this paper we propose a representation scheme for binary images referred to as rectangular coding. This scheme follows the second of the two approaches outlined earlier. It provides a compact representation of binary images and possesses a number of useful properties discussed in the following sections. The algorithm does not result in an optimum solution in the sense of minimizing the number of rectangles. It provides a sub-optimum solution which is shown to be more efficient than several other commonly used techniques including Aoki's algorithm [Fig. 6-1]. Ferrari's algorithm can not be compared with others because it is not applicable to regions with holes.

In describing the algorithm, PASCAL like notations [18] are used in the remainder of this paper.

2. DEFINITIONS AND NOTATIONS

Let 'I' be a binary image produced by digitizing a picture on a rectangular grid.

Definition 1: An object in a binary image 'I' is a 4-neighbour connected region of pixels each of value 1.

The X, Y coordinate system of an image 'I' is defined as follows;

The X-axis is coincident with the uppermost row of the array and the Y-axis is coincident with the leftmost column of the array.

Definition 2: The rectangular code of an object is a set of rectangles which partitions the object completely. Each rectangle is represented by the X, Y coordinates of the upper left vertex, width and height.

The description of each rectangle is stored in the following manner during the process.

```
RECT = RECORD
         X,Y,W,H : INTEGER;
         NEXT : ^RECT;
       END; (* RECT *)
```

where the X and Y fields are the (x, y) coordinates of the upper left vertex of the rectangle, the W field is the width and the H field is the height of the rectangle. The NEXT field is a rectangle pointer which points the next rectangle in the list.

An image 'I' can be represented as a rectangular code of all the objects in it assuming that the background consists of pixels whose value is 0.

Definition 3: A concave vertex is a point on the boundary of an object at which the exterior angle is less than the interior angle.

Definition 4: A chord is a line of finite length which divides a uniform region in 'I' into two regions and is incident on at least one concave vertex.

Definition 5: Any two rectangles A and B are adjacent if and only if there exists a chord which is common to both rectangles as a side or a part of a side.

Definition 6: A closed chain code is a chain code which returns to (ends at) the starting point (the starting pixel) on the boundary.

3. ALGORITHM

3.1. Rectangular Coding

In partitioning an object region into a set of rectangles, the minimum number of rectangles is unique, however, the partitioning of an object region into the minimum number of rectangles may not be unique. In this section we present a heuristic algorithm for constructing rectangular codes that may not provide the optimum solution but provides a compact representation and requires only a single pass through the image.

The process is performed by scanning the array row-by-row from the upper left corner of the image. During the process, this algorithm maintains two lists. One is a list of 'temporary' (growing) rectangles and the other is a list of 'fixed' rectangles. These two lists are referred to as RTEMP and RIMAGE respectively. In scanning a row, runs of 1's are determined. For each such run, a temporary rectangle P is created with the following information:

1. X, Y COORDINATES = X, Y coordinates of the starting pixel of the run.

2. WIDTH = length of the run.

3. HEIGHT = 1.

The list RTEMP is updated with this temporary rectangle P. Consider the process of growing rectangle Q in RTEMP to include the temporary rectangle P, where, 'growing' means the height of rectangle Q is incremented by 1. If rectangle Q is grown, the rectangle P is modified accordingly (width and/or X coordinate of it should be changed), otherwise it is removed from RTEMP and placed in RIMAGE. This process will be continued until all the rectangles in RTEMP are examined or the width of rectangle P becomes zero. Upon completion, if the width of the rectangle P is not zero, P is added to RTEMP. For a given temporary rectangle Q in RTEMP, the following conditions determine whether or not it may be grown to include P (or a part of P).

Conditions :

1. The last row of the rectangle Q is adjacent to the current scanning row.

2. The starting columns of both rectangles P and Q are the same and the width of rectangle P is greater than or equal to that of rectangle Q.

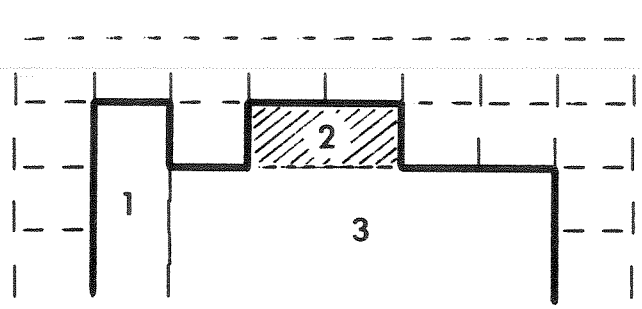
3. The last columns of the both rectangles P and Q are the same and the width of rectangle P is greater than or equal to that of rectangle Q.

For any rectangle Q in the list RTEMP, if condition 1 is false, then the rectangle Q is fixed and moved to RIMAGE, else if condition 2 or condition 3 is true, the rectangle Q may be grown.

These processes are continued until scanning of all the rows in the array is completed. At this point all the remaining rectangles in

RTEMP are fixed, so they should be moved to RIMAGE. RIMAGE contains the rectangular code of the original image.

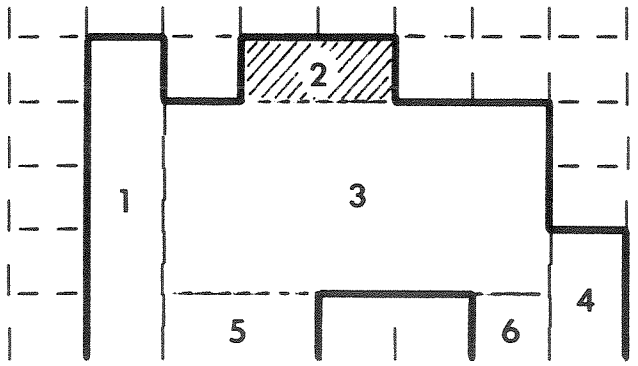
Consider the following example in which the rectangular code for the object region of a sample image is constructed. In scanning the 2nd row, two temporary rectangles 1=(1,1,1,1) and 2=(3,1,2,1) are started, and in scanning the 3rd row, temporary rectangle 1 is grown and a new rectangle 3=(2,2,5,1) is added to RTEMP. At this point rectangle 2 does not satisfy the growing conditions, but it still remains in RTEMP because its last row is adjacent to the current scanning row. On scanning the 4th row, rectangles 1 and 3 grow, and rectangle 2 is removed from RTEMP and inserted in RIMAGE [Fig. 3-1 a]. Actually at the 2nd column of the 4th row, a temporary rectangle P = (1,3,6,1) is created in RTEMP. However with the growth of rectangles 1 and 3, this rectangle P vanishes. After scanning the 5th row, rectangles 1 and 3 are still growing and a new rectangle 4=(7,4,1,1) is added to RTEMP. After scanning the 6th row, rectangles 1 and 4 are growing, 3 is fixed and two more new rectangles (5 and 6) are added to RTEMP [Fig. 3-1 b]. After scanning the 7th row rectangle 3 is moved to RIMAGE and rectangles 1 and 5 are fixed, and 4 and 6 are growing. During the scanning of the 8th row, rectangles 1 and 5 are moved to RIMAGE and rectangle 4 is fixed and 6 is growing. After having completed scanning of all the rows in the array, the remaining rectangles in RTEMP (4 and 6) are moved to RIMAGE [Fig. 3-1 c].



RIMAGE
(3,1,2,1)

RTEMP
(1,1,1,3)
(2,2,5,2)

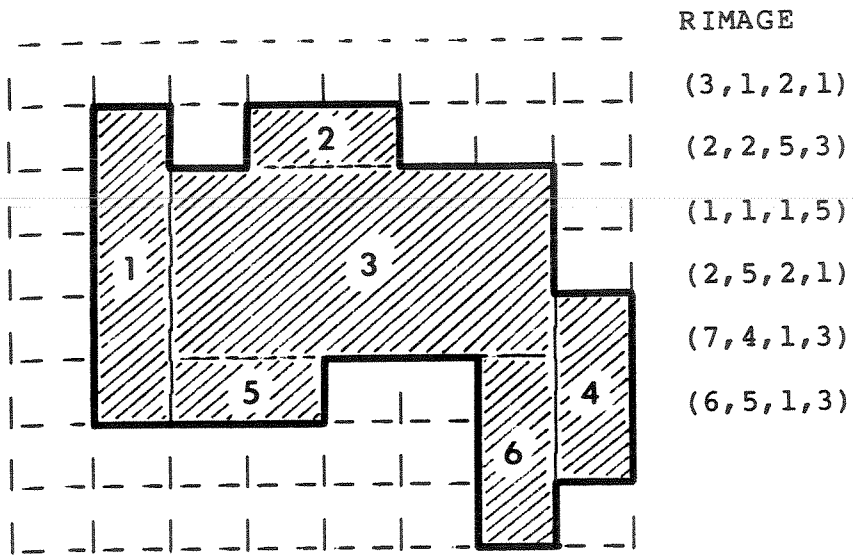
a) After scanning the 4th row.



RIMAGE
(3,1,2,1)

RTEMP
(1,1,1,5)
(2,2,5,3) <--- fixed
(7,4,1,2)
(2,5,2,1) <--- new
(6,5,1,1) <--- new

b) After scanning the 6th row.

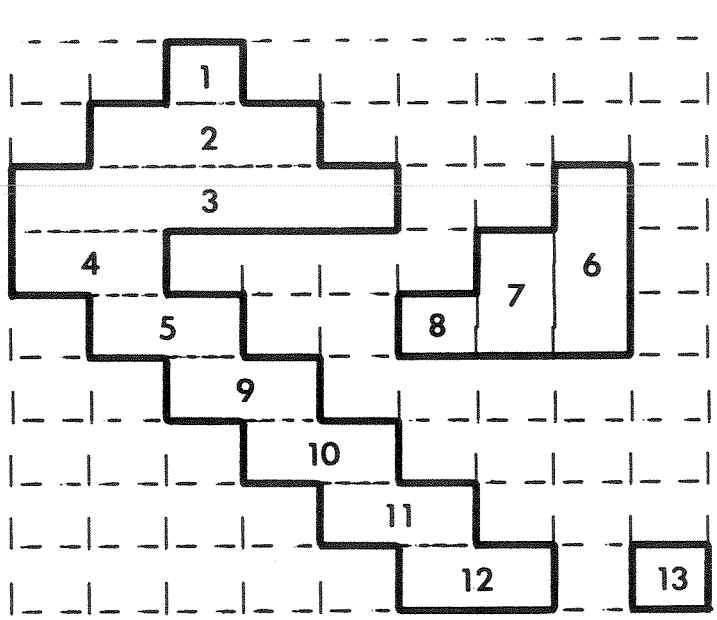


c) Final result.

Figure 3-1: Processing sequence of the algorithm for a sample image.

3.2. Compression of the Rectangular Code

It is possible to exploit certain regularities in the rectangular code and further reduce the storage requirements. Consider Fig. 3-2. Some of these rectangles can be grouped based upon regular spatial relations between them. For example, rectangles 1, 2, and 3 have the same heights and their widths increase symmetrically about X direction in equal increments, and also have the same difference between the X coordinates of two consecutive rectangles in the group. Similar relations are in evidence for the groups (4, 5, 9, 10, 11, 12) and (6, 7, 8). If these relations can be encoded, it is possible to arrive at a more compact representation than that obtained by encoding all the rectangles individually. A group of rectangles will be represented as follows. The first rectangle in a group is selected as a reference



RIMAGE

(2,0,1,1)
(1,1,3,1)
(0,2,5,1)
(0,3,2,1)
(1,4,2,1)
(7,2,1,3)
(6,3,1,2)
(5,4,1,1)
(2,5,2,1)
(3,6,2,1)
(4,7,2,1)
(5,8,2,1)
(8,8,1,1)

Figure 3-2: Regularities in a rectangular code.

rectangle.

```
GROUP = RECORD
          X,Y,W,H,N : INTEGER;
          DX,DY,DW,DH : INTEGER;
          NEXT : ^GROUP;
        END; (* GROUP *)
```

where X, Y, W, H are the parameters of the reference rectangle, N is the number of rectangles in the group, and DX, DY, DW, DH are the differences in each field of consecutive rectangles in the same group.

The number of data elements required to represent a group is greater than that required for two individual rectangles and less than that required for three. Therefore a group has to have at least three

or more rectangles in order for this scheme to be advantageous. The grouped rectangular code for the object regions in Fig. 3-2 is as follows;

```
Rectangle          : ( 8, 8, 1, 1)

The first group    : ( 2, 0, 1, 1, 3,-1, 1, 2, 0)
The second group   : ( 0, 3, 2, 1, 6, 1, 1, 0, 0)
The third group    : ( 7, 2, 1, 3, 3,-1, 1, 0,-1)
```

In this case all the rectangles in a group are adjacent to each other. But it is not necessary to restrict the rectangles in a group to be adjacent to each other. However there should be a systematic way of grouping rectangles efficiently. The grouping should not be arbitrary. We therefore introduce the following constraint in order to make the grouping process systematic.

> In order for two or more rectangles to be grouped together, either the WIDTH or the HEIGHT of the rectangles should be equal.

With this constraint, a group of rectangles can be constructed systematically, and either the DW or DH field of a group code will always be zero. Therefore one integer field may be removed from the group code, and a boolean field can be added to indicate whether it is the height or width that are equal. As a result a group code may be redefined as follows;

```
GROUP = RECORD
          X,Y,W,H,N : INTEGER;
          DX,DY,DS : INTEGER;
          SAMEH : BOOLEAN;
          NEXT : ^GROUP;
        END; (* GROUP *)
```

> where, if SAMEH is TRUE, the heights of rectangles in a group are equal, otherwise the widths are. The DS field corresponds to the difference in width or height of two consecutive rectangles in a group. All the remaining fields are as defined earlier.

This grouping process may be performed in two ways. One is grouping the rectangles after constructing the complete rectangular code of the image and the other is to perform the grouping every time a fixed rectangle is moved from the list RTEMP to RIMAGE. In the case of the former approach, all the rectangles to be checked are already constructed and it can be done by a recursive process. By using this grouping technique we can compress a rectangular code providing a more compact representation.

4. OPERATIONS USING RECTANGULAR CODES

It is often the case that after encoding the image, it is required to transform the original image, e.g., by scaling, translation etc. In this case it becomes necessary to reconstruct the 2-D array from its encoded data structure, apply the required transformation and encode the transformed image. If it is possible to apply these transformations to the rectangular code of the original image, the rectangular code of the transformed image may be obtained directly. This section investigates the types of transformations that can be applied directly to rectangular codes.

Scaling can be performed simply by multiplying the X, Y coordinates, the width and the height of each rectangle by the scaling factor. Translation can be performed also easily by adding the amount of shift to the X, Y coordinates of every rectangle. Rotation by multiples of 90 degrees is not difficult because it does not change the partitions of an object. However, in this case the rectangular code which is constructed from the rotated image may be different from that obtained by rotating the initial rectangular code. Computation

of measures such as area or centroid of objects can be performed easily. The area of the object region is equal to the sum of the areas of all the rectangles, and the center of each rectangle is obtained as follows;

The X-coord. of center = (X-coord. of rectangle)+(Width-1)/2

The Y-coord. of center = (Y-coord. of rectangle)+(Height-1)/2

If an object consists of K rectangles, then the area of the object region is

$$A = \sum_{i=1}^{k} (W_i * H_i)$$

where W_i and H_i are the width and height of the i-th rectangle.

Let the K rectangles of the object, of areas $A_1, A_2, \ldots, A_k$, have coordinates of the centers $(X_1, Y_1), (X_2, Y_2), \ldots, (X_k, Y_k)$, respectively. Assume that the object is in the gravity field, so that the area of a rectangle can be treated as its mass. Then the single force A*g which passes through the centroid of the object has a moment about any axis which is equal to the algebraic sum of the moments of the weights $A_1*g, A_2*g, \ldots, A_k*g$, which pass through the centers of each rectangle. Let (X, Y) be the coordinates of the centroid of the object. Then the X-coordinate of the centroid of the object can be obtained as follows;

$$X*A*g = X_1*A_1*g + X_2*A_2*g + .. + X_k*A_k*g$$

Therefore

$$X = \{\sum_{i=1}^{k} (X_i * A_i)\}/A$$

In similar fashion, the corresponding expression for the Y-coordinate

of the centroid of the object is

$$Y = \{\sum_{i=1}^{k} (Y_i * A_i)\}/A$$

5. ANALYSIS OF THE ALGORITHM

The algorithm consists of two stages; constructing rectangles and grouping them. A single pass through the image is required to construct the rectangles and therefore the processing time for the first stage is of the order of number of pixels in the image. The time spent in the grouping process depends on the number of rectangles checked. If K is the total number of rectangles in the list RIMAGE, the number of rectangles checked is upperbound by K(K-1). This is because the list RIMAGE is already sorted by the row number of the last row of each rectangle and each rectangle is checked with all the rectangles following it in the list. This process is performed twice; once for the 'same height', once for the 'same width'.

In the remainder of this section the storage requirements of rectangular codes are examined and compared with those of other commonly employed representation schemes. This includes examining the increase in storage requirements for the different schemes as the corresponding images undergo a transformation, in this case scaling by an integer constant.

For a 2^n by 2^n image, $2^n * 2^n$ bits of storage are necessary for the 2-D array representaion, and if the size of the image is increased by a factor of N, i.e. to $(N*2^n) * (N*2^n)$, $N^2*2^n*2^n$ bits are required. Therefore the amount of storage required is increased by a factor of N^2.

For a chain code representation, if M is the number of closed chain codes, and L is the total length of chains of the image, the storage requirements will be (2n*M + 2L) bits for 4 - connected chain codes and (2n*M + 3L) bits for 8 - connected chain codes. When the size of the image is increased by a factor of N, the number of bits required to represent a coordinate of a starting point is increased to $n + \lceil \log_2 N \rceil$ and the total length of chains of the image will be NL + (N-1)K, where K is the number of convex vertices of the object. Therefore the amount of storage required will become $2(n + \lceil \log_2 N \rceil)M + 3[N*L + (N-1)K]$ for an 8 - connected chain code.

For a run - length code [17], if C is the number of codewords and L is the average length of a codeword, then L*C bits are required for a given image. If the size of the image is increased by a factor of N, storage requirements increase by O(N). This is due to the fact the number of codewords is increased by a factor of N when the number of rows is multipied by N.

For a quadtree representation [10][12], as the image size increases, there is no need to increase the storage because the storage requirements for a quadtree depends only on the number of nodes of the tree and the number of nodes is not increased even when the size of the image is increased. However if the scaling factor is not a power of 2, it should be stored along with the original quadtree in order to retrieve the scaled image correctly. If the number of nodes of a quadtree is M for a given image, 2M bits of memory are required because we can store a quadtree only with the colors of the nodes if the order of tree traversal is predefined, and the color of a

node can be represented with 2 bits of memory.

Consider the rectangular code. For a 2^n by 2^n image array, the X and Y coordinates, the width and the height of a rectangle can each be represented with n bits. So if the number of rectangles for a given image is K, 4Kn bits of memory are required for this rectangular code. And when the size of an image is increased by a factor of N, the number of rectangles remains same but the number of bits required to represent a field of a rectangle is increased to $n + \lceil \log_2 N \rceil$. Therefore the necessary storage is increased by $4K * \lceil \log_2 N \rceil$ for a rectangular code. But if we use the grouped rectangular code, the storage requirements will be reduced further. For a 2^n by 2^n image array, the DX, DY and DS fields of a group code are bounded by n-1 bits because a group has at least three rectangles in it. Then the storage requirement for this case will be $4Kn + [5n + 3(n-1) + 1]G = 4Kn + 2(4n - 1)G$, where G is the number of groups in the code. Therefore if the size of the image is increased by a factor of N, the number of bits required for each field of the code (except the SAMEH field) is increased by $\lceil \log_2 N \rceil$. So the increase in required storage will be $4(K + 2G) * \lceil \log_2 N \rceil$. These results are summarized in Table 5-1.

Table 5-1: Comparison of storage requirements.

representations	for a given image $(2^n * 2^n)$	for an image scaled by a factor of N	order of incease in storage
2-D array	4^n	$4^n * N^2$	$* N^2$
chain code (8-neigh)	2nM + 3L (note 1)	$2(n+\lceil\log_2 N\rceil)M$ $+3[NL+(N-1)K]$	$+2M\lceil\log_2 N\rceil+$ $3(N-1)(L+K)$
run-length code	L*C (note 2)	L*C*N	* N
quadtree	2M (M is the # of nodes)	2M	(note 4)
grouped rectangular code	4Kn+2(4n-1)G (note 3)	$4K(n+\lceil\log_2 N\rceil)+$ $2[4(n+\lceil\log_2 N\rceil)$ $-1]G$	$+4(K+2G)*$ $\lceil\log_2 N\rceil$

note:

1. M is the number of closed chain codes.
   L is the total chain length.

2. C is the number of codewords.
   L is the average codeword length.

3. K is the number of individual rectangles.
   G is the number of groups.

4. If N is not a power of 2,
   the increase in storage is $+\lceil\log_2 N\rceil$.

6. EXAMPLES

The storage requirements of a rectangular code and some commonly employed representation schemes are compared for a series of sample images in this section. Nine sample images were selected ranging from a comparatively 'simple' image to a 'complex' one [Fig. 6-3 - 6-11]. Fig. 6-1 shows the data reduction ratios of various types of representations in terms of bits per pixel, where the data reduction ratio of one particular representation scheme is defined as follows;

$$\frac{\text{(storage requirements for the representation in bits)}}{\text{(total number of pixels in the image)}}$$

i.e. the storage requirements for a 2-D array is the normalization factor. This ratio should be less than unity for the scheme to be of interest. Fig. 6-2 shows how the actual storage requirements vary when the size of the image is increased by a factor of N. These two graphs indicate that the grouped rectangular code provides a more compact representation of a binary image than the other schemes.

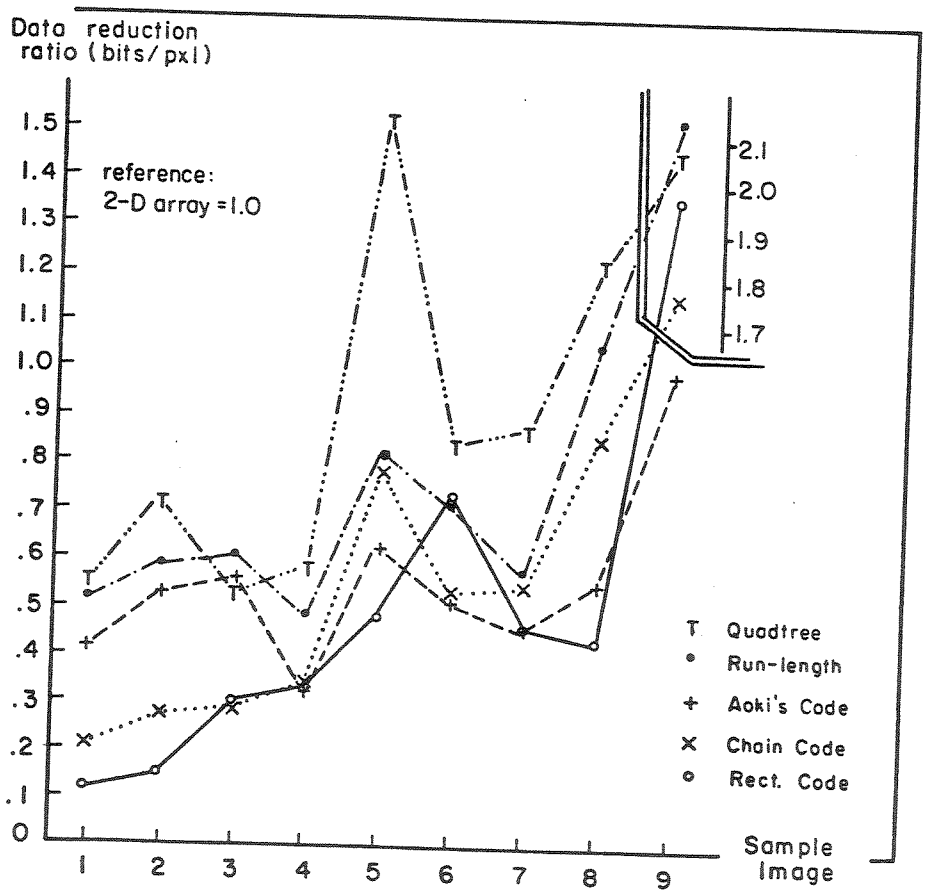


Figure 6-1: Comparison of data reduction ratios

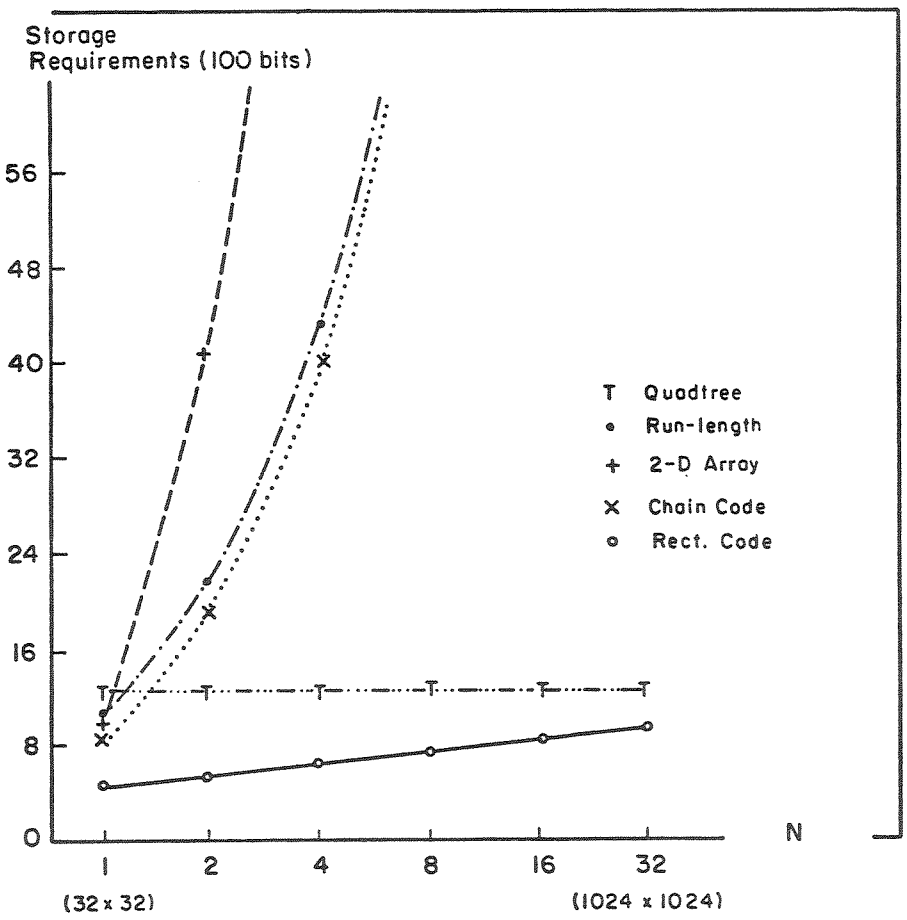


Figure 6-2: Storage requirements when an image is scaled by a factor of N (for sample image 8)

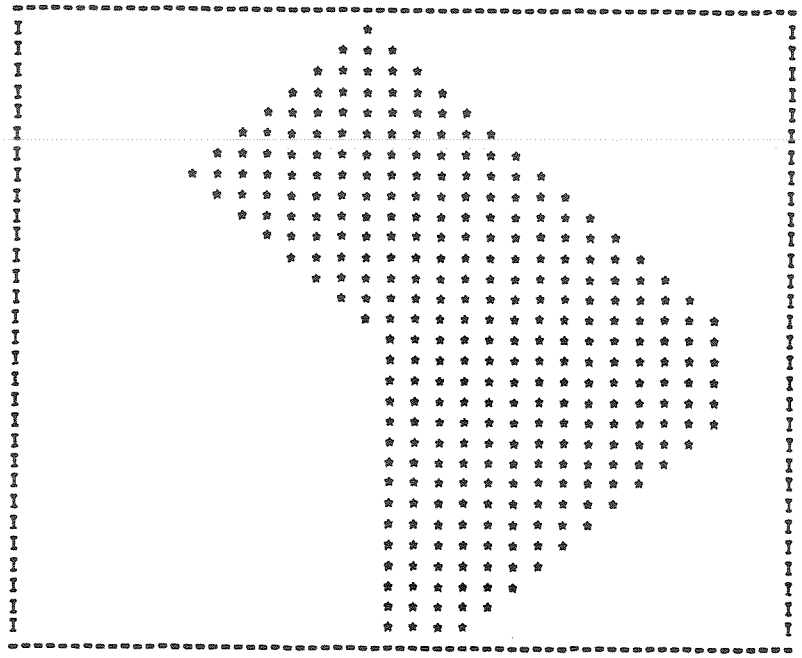


Figure 6-3: Sample image 1

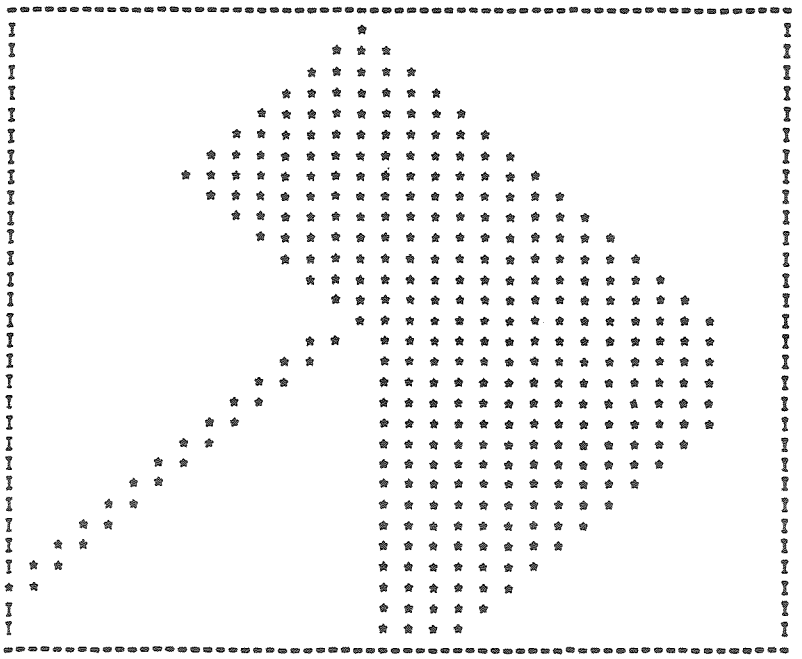


Figure 6-4: Sample image 2

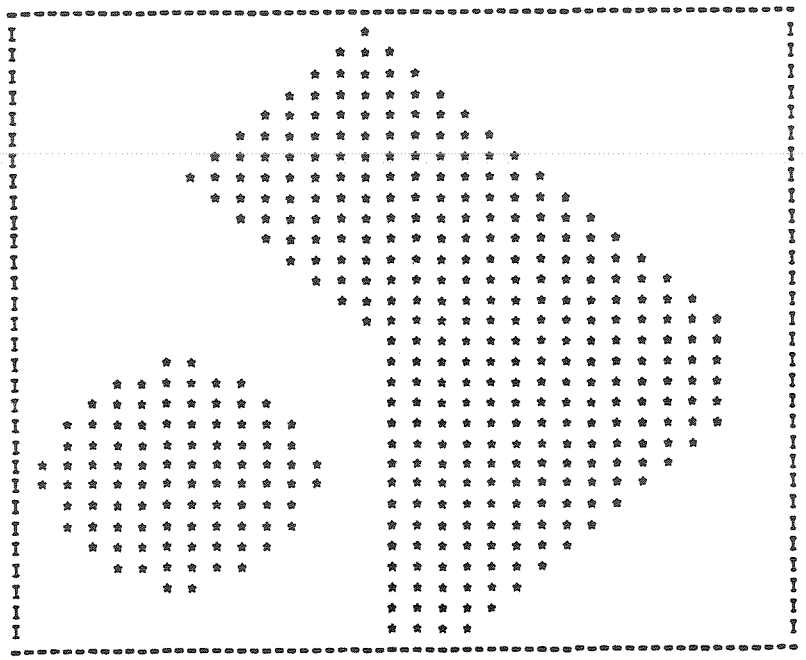


Figure 6-5: Sample image 3

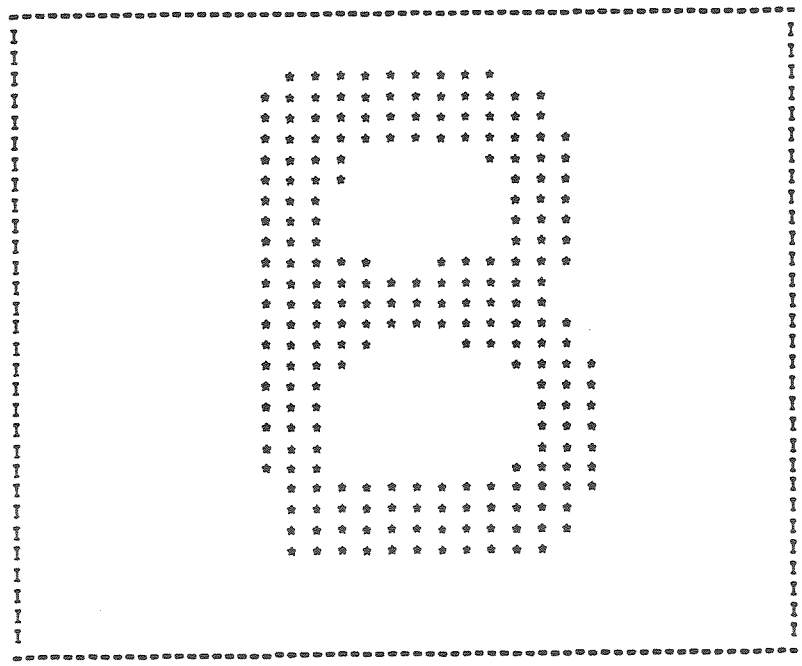


Figure 6-6: Sample image 4

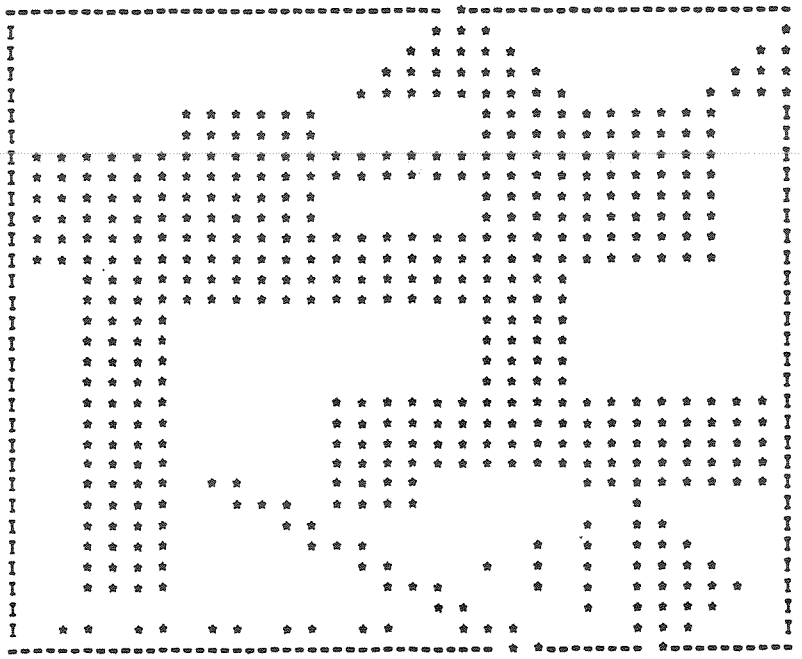


Figure 6-7: Sample image 5

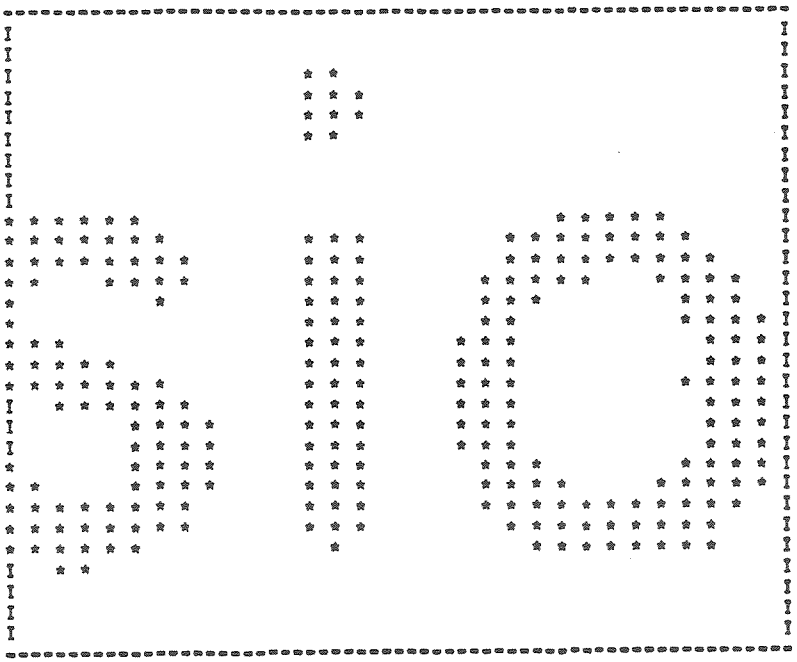


Figure 6-8: Sample image 6

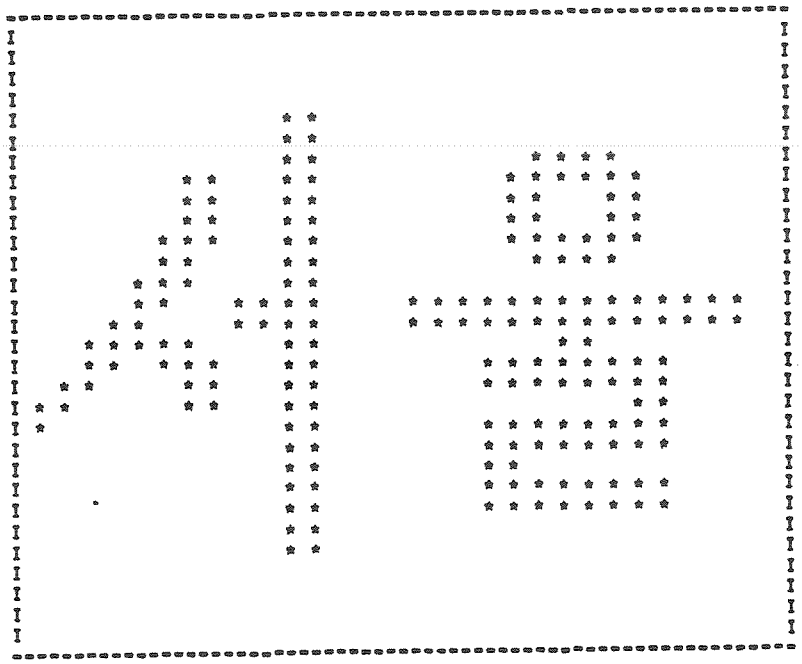


Figure 6-9: Sample image 7

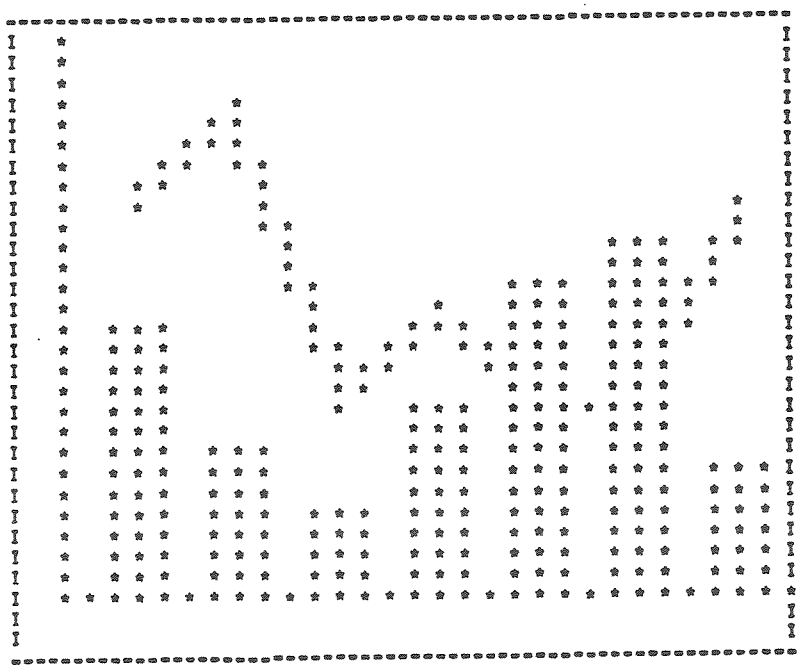


Figure 6-10: Sample image 8

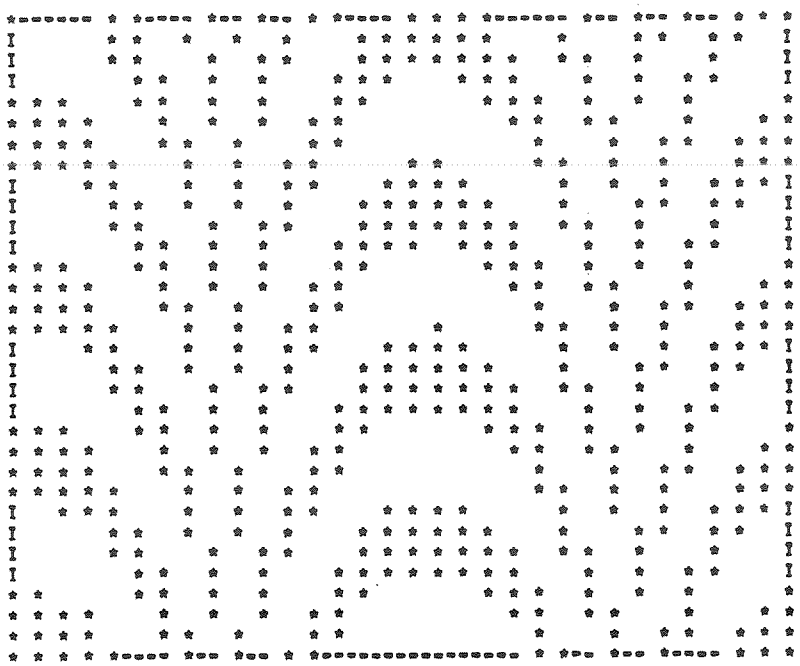


Figure 6-11: Sample image 9

7. EXTENSION TO GRAY LEVEL IMAGES

Gray level images are typically represented as 2-D arrays of integer values each of which represents the gray level of a pixel. The rectangular coding algorithm described in the previous sections deals with only binary images; a special case of gray level images. If a LEVEL field is introduced in both rectangle code and group code, this algorithm can accomodate gray level images. However a few minor changes are necessary for the algorithm to be able to handle gray level images.

Modifications:

1. LEVEL fields should be added in both rectangle code and group code.
2. In scanning a row, find runs of each gray level instead of finding only runs of 1's, and create a temporary rectangle for each run.
3. When the list RTEMP is updated, the following growing condition should also be considered. --- **In order for a temporary rectangle Q to be grown to include the rectangle P, the gray levels of both rectangles must be equal.**
4. During the grouping process, another constraint should be added, i.e., --- **Gray levels of all the rectangles in the same group must be equal.**

This algorithm is more efficient when an image consists of large uniform regions in terms of the gray level. However, in practice, large regions of constant gray level rarely occur. This is due to the fact that as the number of gray levels increase, sensitivity to noise, differences in lighting, shading etc. also increases. Therefore this algorithm may not be efficient for images with a large gray level range. But this algorithm will be efficient when gray level range is small or multilevel thresholding is employed so that an image has a

reasonably small number of gray levels and consists of uniform regions of those gray levels.

This scheme may be applied also to color images. Each pixel in a color image has three primary color components and each component has an intensty value. In other words, a color image consists of three 2-D arrays, one for each primary color. Each 2-D array is same as a gray level image except that it represents intensity values of one of the primary colors. Therefore, by applying this algorithm three times, once for each array, a color image can be encoded as rectangular codes.

An example is shown in Fig. 7-1 and Fig. 7-2. A bar-chart was used in this example. First it was digitized with a gray level range of 0 to 255, and multilevel thresholding was applied. After thresholding the image, the modified rectangular coding algorithm was applied to encode it.

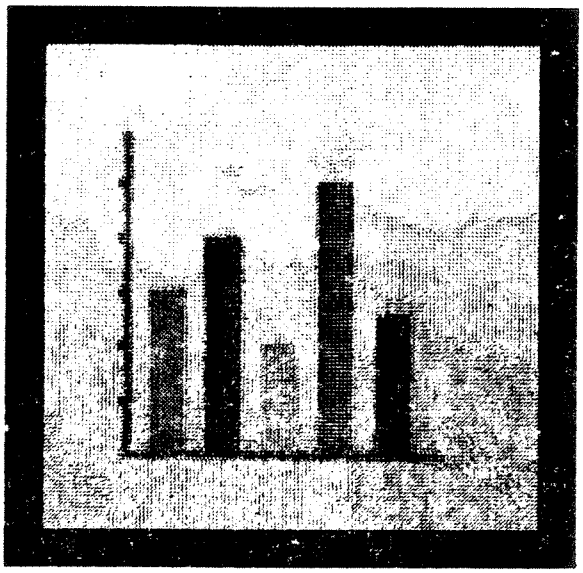
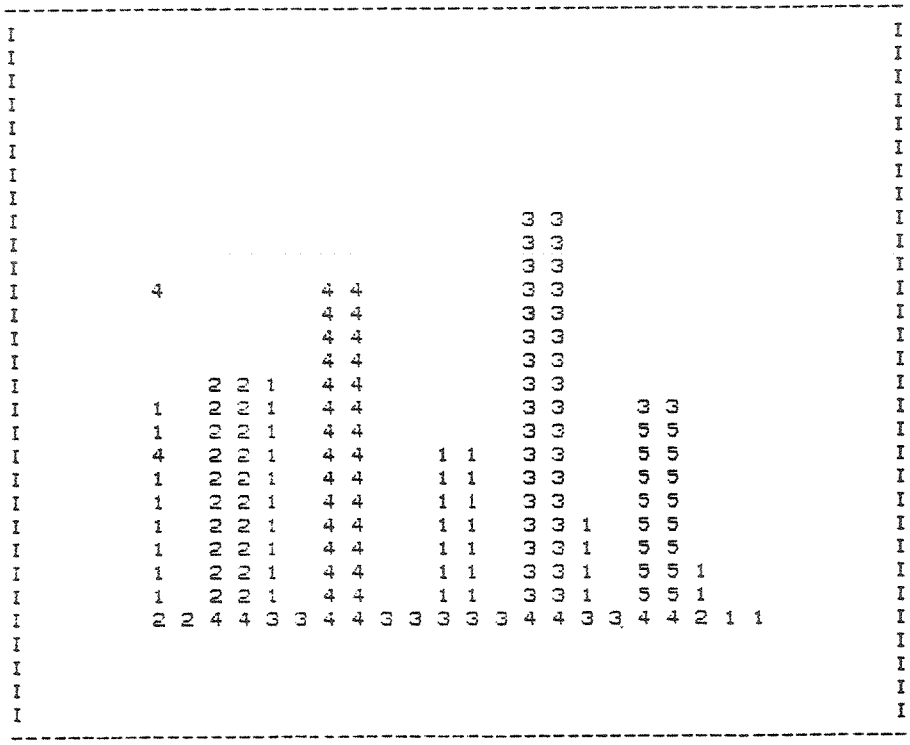


Figure 7-1: Digitized image of a bar-chart



```
1  *** RECTANGULAR CODES OF THE IMAGE ***

(     5    12     1     1     4    )
(    22    17     2     1     3    )
(     5    17     1     2     1    )
(     5    19     1     1     4    )
(    18     9     2    17     3    )
(    11    12     2    15     4    )
(     7    16     2    10     2    )
(     9    16     1    10     1    )
(    22    18     2     8     5    )
(    15    19     2     7     1    )
(     5    20     1     6     1    )
(    20    22     1     4     1    )
(    24    24     1     2     1    )
(     5    26     2     1     2    )
(     7    26     2     1     4    )
(     9    26     2     1     3    )
(    13    26     5     1     3    )
(    18    26     2     1     4    )
(    20    26     2     1     3    )
(    22    26     2     1     4    )
(    24    26     1     1     2    )
(    25    26     2     1     1    )
```

Figure 7-2: Image of the bar-chart after multilevel thresholding and its rectangular code

8. CONCLUSION

An algorithm has been presented for the construction of a rectangular code for binary images. A comparison with several other data schemes indicated that rectangular codes were very efficient, possessing low storage requirements. According to the resulting analysis, this algorithm was found to be especially suitable for images such as bar charts (sample image 8 in Fig. 6-1). The processing time is proportional to the number of pixels in the image. One of the advantages of this algorithm is that each pixel of the image is examined only once during the process, therefore it can be applied to construct a rectangular code directly from a raster scan. It is very simple to perform some basic operations directly on it, such as scaling, translation, rotation by multiples of 90 degrees, and computing the area and the centroid of an object. This is due to the fact that an object is partitioned into a set of rectangles and the position and the size of each rectangle are known. This algorithm can be easily extended to graylevel images simply by adding a LEVEL field in both rectangle code and group code. Improvement in the efficiency of encoding gray level images will be the next problem to receive more attention.

9. ACKNOWLEDGEMENT

We thank S. Yalamanchili, W. N. Martin and A. Mitiche for their many helpful suggestions and advice.

quadtree, IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 1, NO. 2, pp.145-153 (1979)

15. M. Aoki, Rectangular region coding for image data compression, Pattern Recognition, Vol. 11, pp.297-312 (1979).

16. L. Ferrari, P. V. Sankar and J. Sklansky, Minimal rectangular partitions of digitized blobs, Proc. of 5th Int. J. Conf. on Pattern Recognition, pp.1040-1043 (1980).

17. Standardization of group 3 facsimile apparatus for document transmission, International Consultative Commitee for Telephone and Telegraph (CCITT) Recomendation T. 4 (1980).

18. J. Tiberghien, The PASCAL Handbook, Cybex Inc. (1981).