Aggregating raster polygons derived from large remotely sensed images

Steve Barsness
The University of Montana

Aggregating Raster Polygons derived from Large Remotely Sensed Images

by

Steve Barsness

Department of Computer Science, University of Montana

For the Degree of Master of Computer Science

University of Montana

Fall, 1996

Approved by

[illegible]

Chairman, Board of Examiners

[illegible]

Dean, Graduate School

9/20/96

Date

Barsness, Steven J, M.S., September 1996 Computer Science

Aggregating Raster Polygons derived from Large Remotely Sensed Images

[illegible]

Director: Dr. Ray Ford

Remotely sensed satellite imagery produces *rasters* (also called grids or matrices) of numerical *pixels* (or cells). Rasters of raw data are typically processed to form *images*. As sensing technology has advanced, the available rasters have gotten larger and larger, e.g., containing more than 50 million 8 bit pixels. Several types of transformation from raster to image are currently used in various applications, including a class of transformations that identify *raster polygons* (or areas) which represent spatial regions of similar characteristics, as designated by contiguous pixels with equal class values. An important operation in the formation of these images is the aggregation of small raster polygons into larger, adjacent ones. This operation is necessary because small areas may represent "noise", or because the scale of areas may provide information too detailed for analysis in the application domain. This paper examines the algorithmic properties of aggregation, as used in particular applications that create images representing large scale vegetation cover. The paper focuses on two object-oriented implementations which efficiently address the space and time complexity inherent in aggregating large images. Particular attention is given to a novel class of designs created by the author.

Key words: image processing, raster polygon, object oriented design, merge.

Acknowledgments

This project is a synthesis of the work and ideas of many people.

The most important contribution was made by Dr. Ray Ford who originally defined the aggregation paradigm, developed a vocabulary to describe it, and produced the first solutions. He introduced this interesting problem as secondary material in two classes on object oriented design, thus allowing me to contribute my ideas. Other faculty also provided valuable input.

Dr. Alden Wright provided the idea of reinserting areas which change size during the merge process back into a binary heap, thus providing a way to do dynamic reordering which was previously unexplored.

Dr. Ford and Dr. Zhenkui Ma indirectly convinced me that the entire image need not be loaded into memory, thus providing the basis for tBABA, the sliding threshold sized Image band which will likely be the basis for future work on very large images.

Dr. Nick Wilde introduced the breadth first traversal with a queue in his data structures course. This is the foundation for the area traversal method that is the basis of all the ABA variations.

Dr. Roland Redmond and his co-workers provided test data and expert opinion as to the quality of the resultant images.

And of course, the University of Montana Computer Science Department provided the hardware and software platform to develop implementations.

Contents

1 Introduction to Image Aggregation.

The analysis of remotely sensed, digital imagery is important in many diverse applications, ranging from medical diagnosis to ecosystems management. Satellite imagery has become particularly important in ecosystem modeling. Data representing electromagnetic reflectance of the earth's surface provides a way to analyze information about large areas at relatively low cost. The inventory and monitoring of land cover, existing vegetation cover, wildlife habitat, and geomorphic change are just a few current applications.

Remotely sensed imagery consists of datasets collected from rectangular geographic areas called 'scenes'. Each dataset is in the form of a grid of constituent cells or pixels that correspond to approximately square areas on the ground. From a remote platform, data indicating the reflectance measured from within several electromagnetic (EM) bands are assigned to each corresponding cell. A typical cell size, as used in the work here, corresponds to a 30 meter square area on the ground. Other sizes, both larger and smaller, are also common [10].

Data can be collected from several different EM bands by more complex sensors, then numerically combined to produce a single value for each cell. **Classification** is a basic operation which combines the raw sensor data for each cell along with other information about the cell to determine a cell class member value. Class membership is usually represented by a single integer drawn from a fixed range, usually smaller than the original data range. For example, 24 bits of raw data per cell can be classified into an 8-bit value representing membership in one of up to 256 disjoint sets. Classification may be directed by a problem domain specialist, which is called **supervised classification**, or it may directed automatically by the properties of the data itself, which is called **unsupervised classification**. Once all cell values have been classified, the rectangular collection of classified data values is called an **Image**.

In this paper, we are concerned with processing that involves classification of large datasets to produce large images. Large, by today's standards, means images with more than 50 million 8 bit pixels (e.g., a 7500 x 7500 grid). The large number of pixels in such images forces us to look for higher level entities in the image, which we call **raster polygons** or simply **areas.**

As a simple explanation of areas, consider assigning a unique color (a false color, not the actual color) to each classified data value in an image, then displaying the result on a graphics monitor. Areas with the same color stand out as polygon shaped regions. The goal in our analysis is to identify these areas, then to refine the image to reduce the large numbers of single pixels or areas smaller than some threshold size that are typically present.

Intuitively, areas represent contiguous collections of pixels with common properties. In the application considered here, the property of interest is land cover. Large images that contain a myriad of small areas yield databases of impractical size, which greatly complicates analysis and/or corresponding 'ground truthing' of the dataset. Thus, in addition to simple classification, it is also useful to transform large images with many small areas into images with aggregated larger areas, thereby reducing the data volume to an manageable level. Our work focuses on image aggregation which is based on area size and other attributes. We assume that a **minimum mapping unit (MMU)** defines the minimum acceptable size of a remaining area. We explore aggregation rules based on the 'similarity' between areas; we want to aggregate a too-small area with the 'most similar' big-enough neighboring area. The University of Montana Wildlife Spatial Analysis Laboratory is concerned with applications in which aggregation governed by rules that describe relationships between classified data values and high-level entities. Such an approach treats aggregation as a distinct process that follows classification, i.e., operating on classified data values, not the originally sensed data values.

We describe in detail two designs for implementing aggregation efficiently for large images. The designs are referred to as row-by-row area bounded (RBR) and area-by-area (ABA). We examine one particular implementation of RBR and three implementations of ABA: ABA with static ordering (sABA), ABA with dynamic ordering (dABA), and ABA with a sliding threshold band (tBABA). Prior to discussing these algorithms, we review other similar types of aggregation processes that have been developed elsewhere, then formally define the aggregation problem that we are solving.

2 Related Work

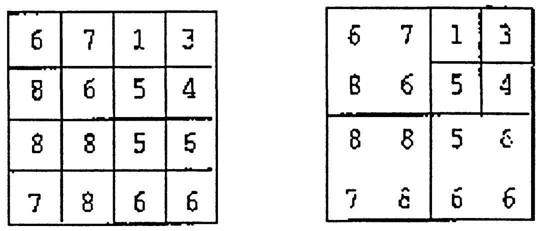
2.1 Syracuse University's Region Growing

Researchers at Syracuse University have developed a technique which they refer to as region growing [3]. Heterogenous regions, composed of differing pixel values, are systematically grouped into regions exhibiting greater homogeneity according to some homogeneity function H(R). As illustrated in Figure 2.1, H(R) is a boolean function that evaluates true when the region in question exhibits a range between the minimum and maximum that does not exceed a threshold value T.

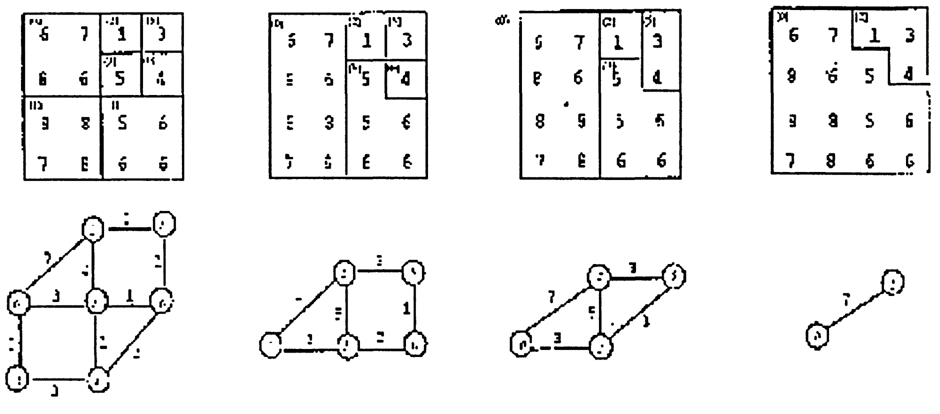
Syracuse's region growing proceeds in two stages, the split stage and the merge stage. The split stage is described as follows: "At first, each pixel is considered a homogeneous square region of size 1x1. Then every group of four adjacent pixels is tested for homogeneity. If the homogeneity criterion is satisfied, the four square regions are combined into one larger square region of size 2x2. Next, every group of four adjacent square regions of size 2x2 is tested for homogeneity. If the homogeneity criterion is satisfied, the four square regions are combined into one larger square region of 4x4, and so on... The split stage terminates when the whole image is one square region size NxN, or when no more square regions can be merged." [3]

The merge stage is described as: "The merge is achieved by reformulating the region growing problem as a weighted, un-directed graph problem, where the vertices of the graph represent the regions in the image, and the edges represent the neighboring relationship between these regions." [3] At each iteration in the

$$H(R) = \begin{array}{l} \nearrow \ \textit{true}, \text{ if for all point pairs } (x_1, y_1) \text{ and } (x_2, y_2) \text{ in } R, \\ \qquad \|f(x_1, y_1) - f(x_2, y_2)\| < T. \\ \searrow \ \textit{false}, \text{ otherwise.} \end{array}$$



Square regions: (a) at start of the split stage, (b) after first and final split iteration



Regions: (a) at start of the merge stage; (b) after first merge iteration; (c) after second merge iteration; (d) after third and final merge iteration

Region Growing: top, the homogeneity function; middle, the split stage with a threshold of 3; bottom, the merge stage. [3]

Figure 2.1

merge stage, a region merges with the neighbor that best satisfies the homogeneity criterion. The merge stage continues as long as it can select neighbors that satisfy the homogeneity criterion.

Region growing is essentially a form of unsupervised classification that looks only to set individual pixel values. The technique produces images that are very different than ours. It doesn't derive the final image from the properties of homogeneous areas containing only one class value, but instead uses heterogenous regions containing many similar class values.

Syracuse has implemented a parallel algorithm to perform region growing, running on Connection Machines CM-2 and CM-5. It is difficult to compare region growing with our aggregation technique. In the general case, H(R) is O(n^2) where n is the number of pixels in R. Every pixel in R must be compared to every other. In our aggregation, the manipulation of comparable 'areas' is O(n).

2.2 California's Similarity Filtering

Another aggregation technique comes from California's Department of Forestry [2], described as 'similarity filtering'. This filtering technique involves passing a 3x3 'window' over each pixel, and then determining the value of the central pixel as some function of its eight neighbors. A wide range of selection functions can be considered. Most easily understood is modal filtering, illustrated in Figure 2.2, which chooses the most numerous value in a 3x3 window as the value of the central pixel. California's technique involves use of a more complex selection

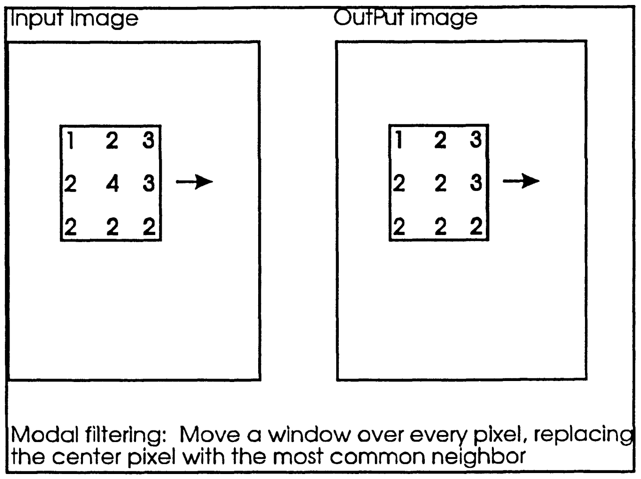

Input Image
OutPut image
1 2 3
2 4 3
2 2 2
1 2 3
2 2 3
2 2 2
Modal filtering: Move a window over every pixel, replacing the center pixel with the most common neighbor

Figure 2.2

function designed to select the most **similar** neighbor.

Assuming a small filtering window which can be analyzed in constant time, similarity filtering will be faster than our aggregation technique that is based on areas, i.e. strictly O(N^2) where N is the dimension of a square image. However, as a pixel by pixel operation similarity filtering suffers from area boundary distortion that is common in all filtering techniques. It is useful in some types of image enhancement, but not area formation.

2.3 ESRI's 'eliminate and nibble'

Two additional aggregation techniques that are commonly used are ESRI's [1][6] 'eliminate' and 'nibble' functions. These are operations implemented in the software package Arc/Info, and thus are attractive because they are preprogrammed and relatively inexpensive to perform. 'Eliminate' is a vector technique where the small area to be eliminated is dissolved into the neighbor with which it has the longest common border. 'Nibble' is a raster technique where neighbor selection is computed point by point via a minimum Euclidean distance applied to each point data value. Neither technique performs aggregation directed by user-specified similarity criterion. Neither technique supports automated processing based on the concept of a minimum mapping unit, i.e. a human operator must identify and eliminate/nibble all small areas away one by one.

2.4 Tennessee's Area Identification

Work at The University of Tennessee [4] focuses on the same sort of area identification we do, but does no aggregation. They describe recursive and pseudo-recursive area identification using a stack, building areas incrementally as they are encountered; this is the same as the area identification method used in our RBR. Most of the focus in [4] is on the implementation of a parallel algorithm for area identification suitable to run on a Connection Machine (CM-5). The performance results reported in [4] describe relatively small artificially constructed images, so it isn't clear how their approach performs in practice, with large images containing a real mix of large and small areas.

3 Montana's Aggregation Paradigm

3.1 Image aggregation Overview

The starting point for our approach to aggregation is that areas are identified in a classified image, each small area will be moved in toto into a 'target' neighboring area, that this process will start with the smallest areas, and continue until all the areas in the image are as large as a specified **threshold size** that defines the **MMU** (threshold + 1 = MMU). Thus the fundamental unit of aggregation is an area composed of classified pixels, not a filtering window.

Conceptually, the aggregation process can be described in four steps:

1) <u>input</u> the original image, threshold size, and other problem parameters;
2) <u>identify</u> all the areas in the image and <u>partition</u> into disjoint sets containing to-be-merged areas (TBMs) and the survivor areas (SURV);
3) <u>process</u> each TBM <u>in</u> a specified <u>order</u>, first identifying the target neighbor (TN) that will receive the TBM's pixels, and then effect the <u>merge</u>, which modifies the image to delete the TBM and expand the TN;
4) <u>output</u> the final image which will contain only areas as large or larger than the threshold.

To elaborate, consider aspects of this process in more detail.

3.2 Area Identification

In 'area identification', the word 'identify' is used loosely. Identification must somehow record areas, but the extent to which information is saved to facilitate future access to the area (e.g. to read or change its pixel values, or lookup size and location) can vary widely.

Identification must be based on pixels which have the same data values as their 'neighboring pixels', but there are several potential definitions of 'neighbor'. Here we use the 'NEWS' definition of neighboring pixels, an acronym for North, East, West, and South which in Figure 3.1, considers only orthogonally adjacent pixels to be neighbors. Other definitions of neighboring pixels such as those that include 'diagonal neighbors' are possible, but are not used here.

The result of identification is the formulation of a set of areas, each which includes only neighboring pixels of the same data value. With 8-bit data, pixel values vary between 0 and 255. However, there may be millions of areas, and each requires a unique area identifier. Conceptually, identification can also imply construction of a list (possibly ordered) of those identifiers which reference an area descriptor that stores other attributes for each area, and thus facilitates future lookup of the properties of a given area.

3.3 Partitioning and ordering

Given area identification, partitioning into TBMs and SURVs is simple. The user specifies an area threshold size that can be used as the basis for partitioning; additional special cases, such as designated 'no merge' data values can also be specified. Since partitioning is based on size, it is natural (but not mandatory) to save each area's size as it is identified.

The determination of proper merge ordering is a more complex issue. Objectively, there is no 'right' or 'wrong' order. If there is a correct order to merge the TBMs, it is application-specific. TBMs could be merged randomly, from smallest to largest, from largest to smallest, or ordered based on some other property. Here we consider three different ordering schemes: arbitrary ordering (order based on an algorithm-specific 'first encounter'), smallest to largest with the size fixed during area identification (static ordering), and smallest to largest with the size based on current area size (dynamic reordering). Each produces different resultant images.

3.4 TBM Mergers

Once an appropriate order is determined, we can address the details of processing a particular TBM, i.e., we find its appropriate TN, then effect the merge. A TBM will have one or more adjacent areas, so some function of the neighboring areas will be used to select the TN. The merge itself can be accomplished by modifying pixel values in an image, modifying an area descriptor, or both. Figure 3.2 illustrates replacing the TBM's pixel's values with the pixel value of the TN. Following processing of all TBMs, the image may need to be reconstructed (i.e. if descriptors only are changed), before the final aggregated image is output to disk.

Within this simple paradigm there are two additional subtle aspects to consider. The aggregation process changes the size and other attributes of areas as it progresses, e.g., each merger increases the size of the TN. For orderings that are based on TBM area attributes, this raises the question of exactly how/when the order is computed. This is the distinction between static and dynamic TBM ordering. In another special circumstance, when an area is merged, one or more disjoint neighbors can have the same data value as the TN. Following the merge, we can end up with two neighboring areas that have the same data value, and thus could also be merged, We refer to this as the **cascade merge** problem. Figure 3.2 illustrates the cascade problem. There are two TNs with a class value of 5; either or both of these could be merged with the TBM. Other examples could have more than two candidate TNs.

Even with a specific application, the choice of merge order and the treatment of how to handle cascades may be arbitrary, and different choices may yield different final images. To achieve a fully deterministic, implementation-independent result, many choices must be made rigidly by convention. However, within the constraints of the hardware platforms and the chosen algorithm, seemingly arbitrary choices may be easy to implement in some situations, and impossible in others.

The following is a list of conventions with which our implementations and performance test comply:

1. We use the 'NEWS' definition of neighboring pixels, extended to include both a pixel's neighbors and an area's neighbors.
2. We define areas greater or equal in size to the threshold as survivors.
3. We generally assume that merge order proceeds from smallest to largest with ties being decided by lexicographic order, i.e. the North-most-West-most point in the area. However, we also consider variations on this theme in implementations that reflect arbitrary ordering, static ordering based on original size, and dynamic ordering based on current size.
4. We assume that no cascade processing is done, i.e., that neighboring same-valued areas produced by a merger are left as separate areas.

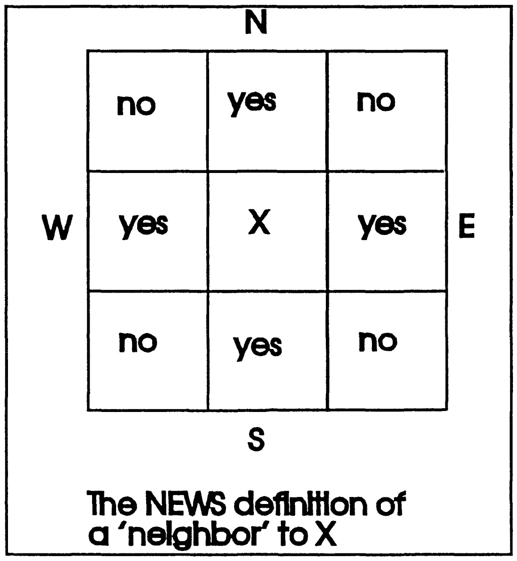
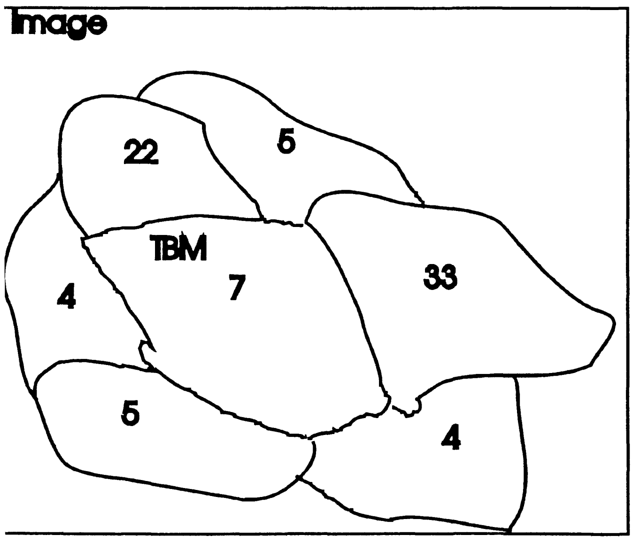
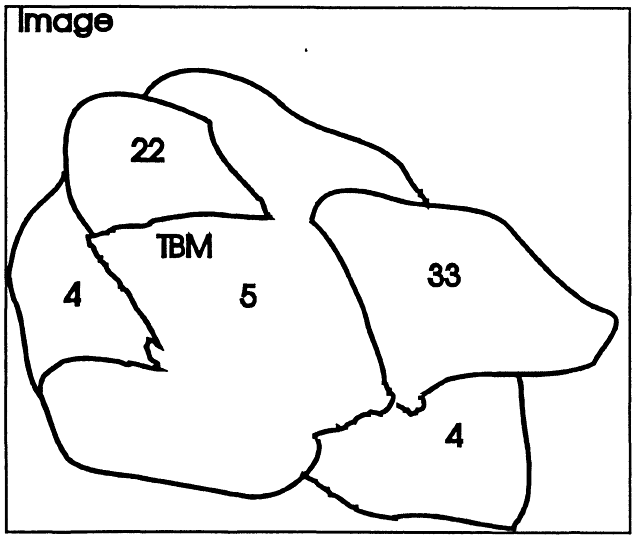


Figure 3.1



Areas before a merge



The same group of areas after a merge. The neighbor most similar to the TBM had an image value of 5.

Figure 3.2

4.0 Design Implementation Comparisons

4.1 Object Orientation

To designs have emerged to implement the Area Aggregation paradigm. The first was done by Ray Ford in 1993 [1], in a design referred to as row-by-row area bounded (RBR) and the basis of a number of implementations [1,6,8]. Subsequently, in 1996 a group of Ford's students--myself and Dale Hamilton--produced an alternative object oriented design and implementation which will be referred to as area-by-area processing (ABA). In all studies of aggregation for large imagery, it is obvious that internal memory demands impose a real constraint on the implementation. Therefore, the primary goal of both RBR and ABA designs is to minimize memory utilization; a secondary goal is to achieve fast run times. Early implementation of RBR demonstrated that is possible to process large images (e.g. 8000x8000) on simple workstations; later versions of RBR and ABA have dramatically reduced both memory and time costs.

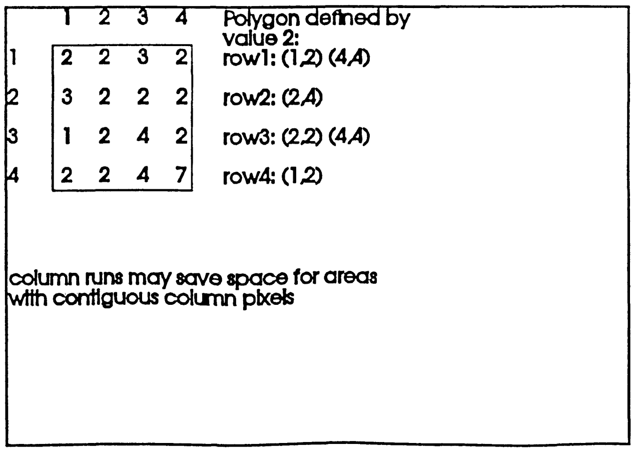
To highlight the similarities and differences between the two designs, functional descriptions of the constituent objects can be used as a basis of comparison.

RBR's objects:	ABA's objects:
Image	Image
AreaContainer	AreaContainer
Area	Area
AreaFinder	AreaFinder
AreaMerger	AreaMerger
Point	Point
SimilarityTable	SimilarityTable
NoMergeTable	NoMergeTable
TargetSelect	TargetSelection
PointSet	BitMap
NeighborList	PointQueue

Figure 4.1

4.2 RBR vs ABA Comparisons

The underlined objects in Figure 4.1 highlight the central differences between the two designs. In both designs, a **Point** is a column-row coordinate pair referencing a pixel in the Image. RBR's **PointSet** is conceptually a collection of points that refer to equal and adjacent values in the image. Each Area contains exactly one PointSet. There are various ways that an implementation may describe an Areas's Points, e.g. a list of points, or a list of column runs as illustrated in Figure 4.2. Generally, complex encodings allow large areas to be represented with relatively few bytes of storage, however, such encodings can be cumbersome to decode and encode when **PointSets** are merged. All existing versions of RBR represent Pointsets as autonomous, explicit objects in some manner.



The RBR represents PointSets as column runs which can save space in large areas.

Figure 4.2

As an alterative representation, ABA uses the **Image** itself to avoid saving explicit pointsets. ABA's references are to actual components of the image. The advantage to RBR's autonomy is that PointSets may be discarded when they are no longer needed, as would be the case when an Area becomes a survivor. ABA's pixel collection resists autonomy because, in the Image, areas exhibit mutual definition. For example, if one area's pixel value is changed to that of its neighbor, this must change the size and shape of one or more areas. Area representations can never be discarded.

Another major difference between the two designs' objects is the determination of an Area's neighbors. All versions of RBR maintain a discrete object for each area called a **NeighborList**. The NeighborList may be discarded when an area becomes a survivor, but it must be kept 'current' in the course of other merges as long as an area is smaller than the threshold. ABA determines an Area's neighbors when needed, during merge target selection, by re-examining the TBM in the current image. During re-examination, ABA uses a **PointQueue** to traverse the TBM, and a **BitMap** to ensure that each pixel in the TBM is visited only once. Image values different from the TBM's are considered as candidate for target neighbor. Such a determination of neighbors is often referred to as 'on-demand' processing, vs. the 'lookup' provided by an explicit NeighborList.

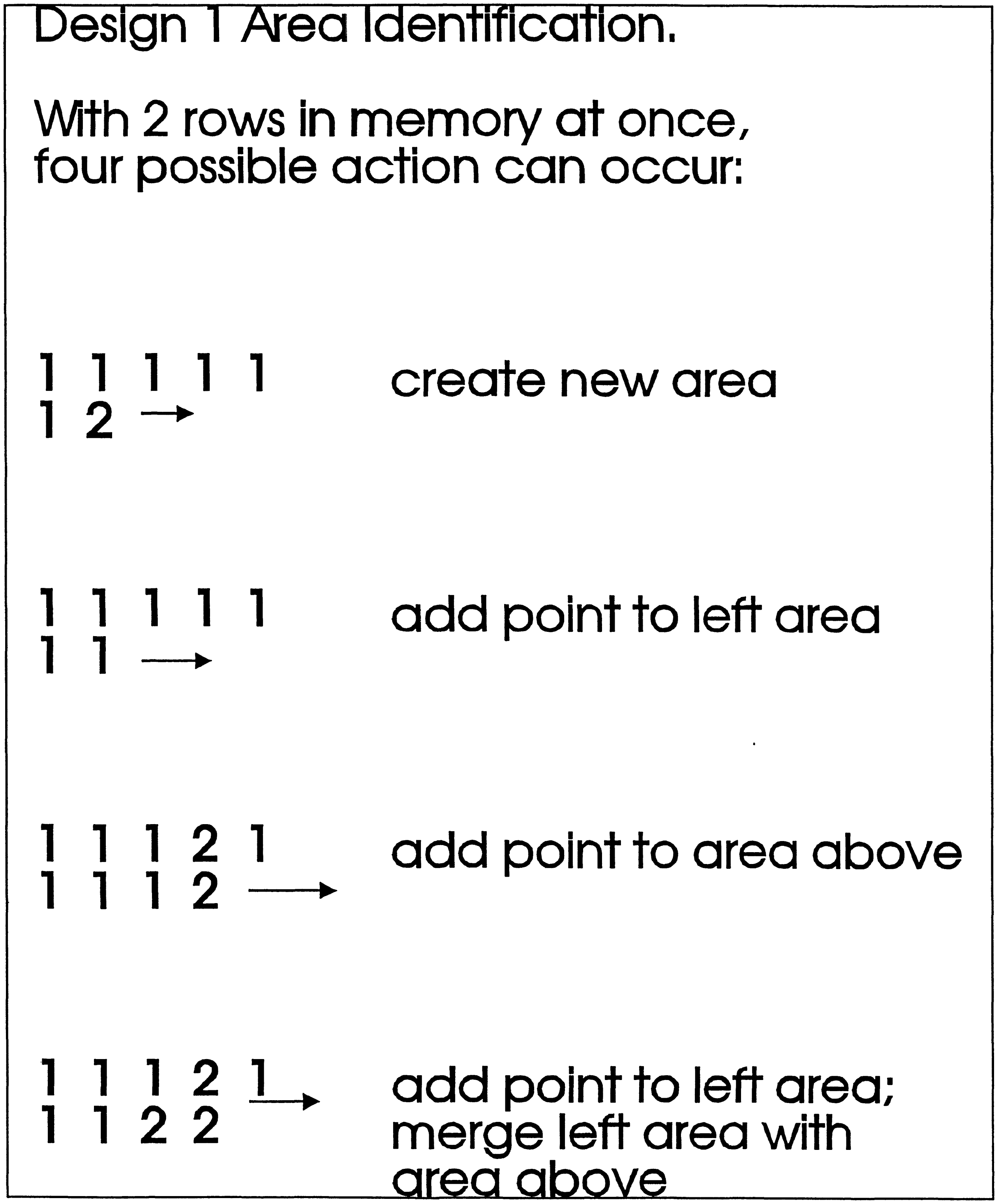
The other objects in both designs have similar functionality, and therefore are named the same, but because they are acting on central objects which are fundamentally different, their specifications vary slightly and their implementations vary greatly.

The **Image** in both designs is similar; both view the image as a byte matrix. The RBR approach is concerned with exploiting the maximal amount of information available from each new row in order to avoid having to store the whole image in memory at any time. Thus, after reading a new row, RBR attempts to do significant Area identification and possibly merging. Therefore, a basic behavior of RBR's AreaFinder is to retrieve one row from the Image, then process it before moving to the next row.

Early versions of ABA (sABA and dABA) read the entire Image into memory, select a pixel value, and 'follow it around' to identify an area, thus accessing the image randomly. However, later versions incorporate some aspects similar to row-by-row processing to avoid holding the entire Image in memory at once. Both RBR and ABA designs can be adapted to process sequences of rows representing horizontal 'bands' of the image as a group, rather than single rows. While reading the image sequentially at row R, only T+1 (threshold+1) rows below R are necessary to identify TBM's and only T+1 rows above R are necessary to merge the TBMs. Thus a relatively narrow band of the image needs to be in memory at any one time. Band processing is in the RBR implementations of Guo [6] and Ma [8]. Our implementation of this threshold band method is referred to as tBABA.

In both designs, Area identification is accomplished by an **AreaFinder** object, but implementations vary greatly. The RBR design holds at least two rows from the image, scans the current row one pixel at a time, left to right, and compares the current pixel value to the values in the pixels to the left and above (in the previous row). Then one of four actions is taken, as illustrated in Figure 4.3: 1) a new Area is created and the point is added to the Area's PointSet; 2) the current point is added to the Area (and PointSet) to the left; 3) the current Point is added to the Area (and PointSet) above; 4) the left and above areas are merged and the current point is added to the resultant area. The status of an area is updated each time its size changes. At the end of each row the set of **bounded** areas can be identified, where a known area that has no pixels in the current row must be **bounded**. Once an area is bounded, it can be determined to be a TBM or SURV, and can be processed on-the-fly. Thus, by the time the last row in the Image is processed, the area identification, and possibly the merge, are complete.

ABA's Area identification is completely different. After the entire Image (or partial band) is read into memory, all bits corresponding to pixels in a BitMap are initialized to ones. The BitMap has the same dimensions as the Image. In the Image, an area can be thought of as a graph, where neighboring pixels with the same image value are considered to be neighbors or children (after an order has been imposed on the graph) of a parent pixel (see Figure 4.4). As such, an area's pixels can be traversed (visited) as a graph using a breadth-first-traversal. Note that without modification, the matrix representation of the Image is sufficient to express a set of graphs. Neighboring pixels can be calculated by simply adding or subtracting 1 from a given point's coordinates. A **PointQueue** (a queue of points) is used to facilitate the breadth-first-traversal, but explicit graphs based on pointers are not needed (see Figure 4.5).



Design 1 Area Identification.

With 2 rows in memory at once,
four possible action can occur:

```
1 1 1 1 1      create new area
1 2 →

1 1 1 1 1      add point to left area
1 1 →

1 1 1 2 1      add point to area above
1 1 1 2 →

1 1 1 2 1 →    add point to left area;
1 1 2 2        merge left area with
               area above
```

RBR Area Identification

Figure 4.3

ABA scans the image in row-by-row, column-by-column order. Upon finding a pixel which is marked 'unidentified' in the **BitMap**, it traverses the Image's pixels in breadth first order, as illustrated in Figure 4.6. As the traversal proceeds, when a pixel is visited the corresponding bit in the **BitMap** is replaced with a zero. Only the NorthWest Point of the area and the area size are stored as an Area attributes in the **AreaContainer**. The NorthWest point is used both as a unique area identifier and as a point of reentry into the image for subsequent traversals. When the breadth first traversal ends, ABA returns to the image to look for a new area. When this scan finishes, the **BitMap** will contain all zeros indicating that all areas have been identified.

During area identification in RBR, as each area is bounded it is labeled as a TBM or SURV, then bin-sorted by size into T (threshold) bins, labeled as a TBM or SURV. Thus, RBR needs no explicit sort after area identification is completed. sABA's and dABA's uses a binary heap to order TBMs with each identification via insertion into the heap. ABA excludes survivors from the heap because they are not needed in ABA's merge. tBABA bin-sorts TBM NorthWest Points into T binary heaps, with NorthWest sort order maintained in each heap.

After ordering, in either design, the **AreaMerger** object is activated. The merge order depends on Area size and location. Merging proceeds smallest to largest, and among areas with the same size, the least NorthWest coordinate in lexicographic order determines sort order. Sort order is based on the original size of an area in RBR and the static sABA implementation; the dynamic dABA implementation reinserts areas into the heap as the merge progresses and area sizes change.

In the chosen order, for each TBM area, RBR uses its NeighborList to examine the characteristics of its neighbors. By some function of similarity between neighboring Image values, a merge target neighbor (TN) is chosen. The TBM contributes its points to the TN and the TBM is deleted. However, before the merge is complete, all the TBM's neighbor's **NeighborLists** must be logically or physically updated to replace references to the TBM with references to the TN. RBR's Image modification is accomplished by using each area's **PointSet** to reconstitute the image at the end of the whole merge process.

ABA similarly selects TBMs in specified order and uses a similarity function, but selects candidate TN pixel values by examining neighboring pixel values by doing a breadth-first-traversal of the TBM in the current Image. As the traversal progresses, the 'most similar' neighboring pixel·value is saved. The actual merge is accomplished by once again doing a breadth-first-traversal of the TBM, to modify TBM pixels to their new value. This process requires multiple passes through parts of the image for each TBM (survivors are examined only once), but does not involve **PointSet** mergers, **Neighborlist** updates, nor **Image** reconstitution because the image is modified directly.

In the dynamic reordering version dABA, when the first traversal progresses, the size of the TBM is calculated. If the TBM's original or previous size has not been changed, the TBM is merged; otherwise, it is reinserted into the **AreaContainer's** heap and merged later, when the TBM is re-encountered and exibits a stable size.

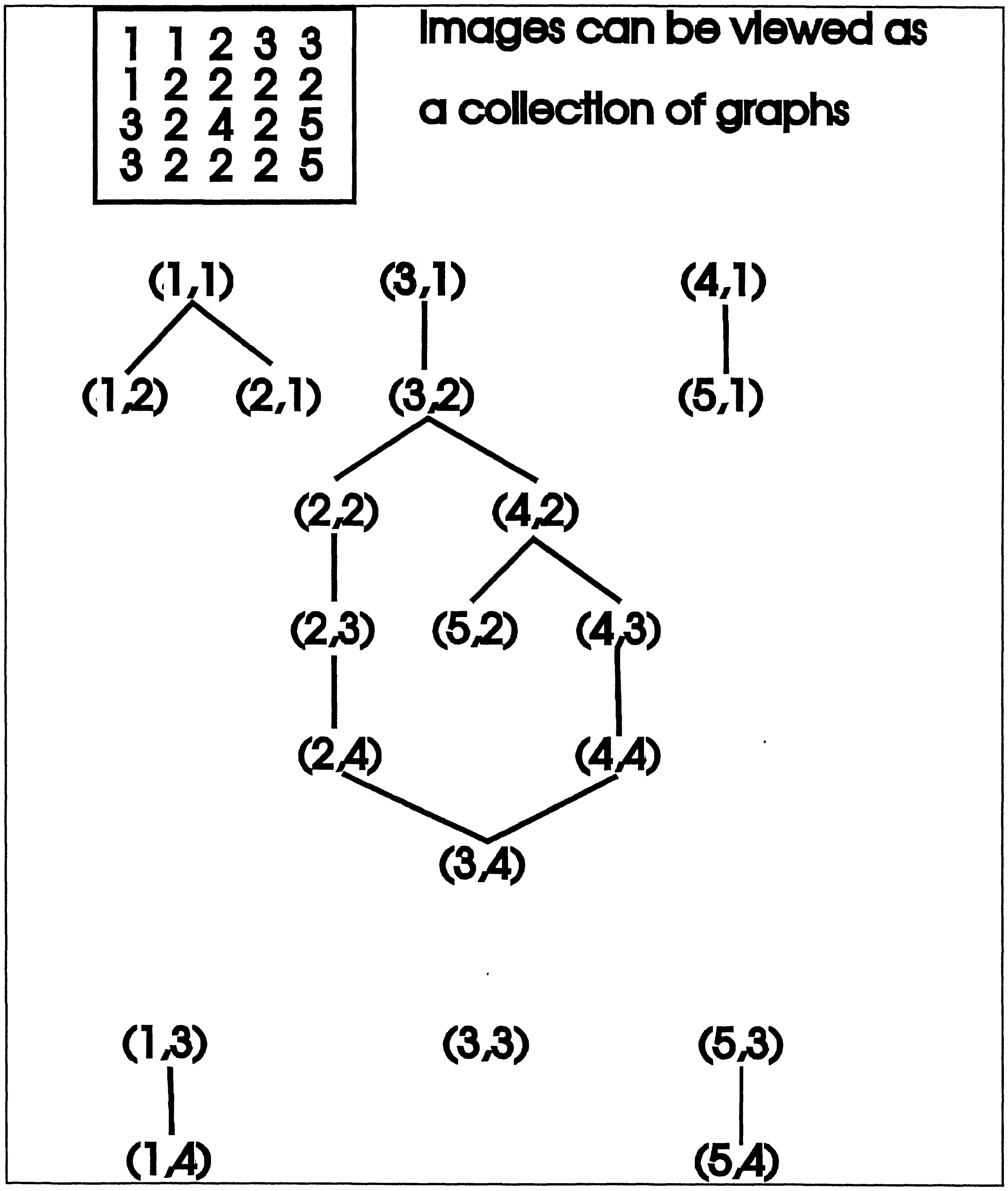
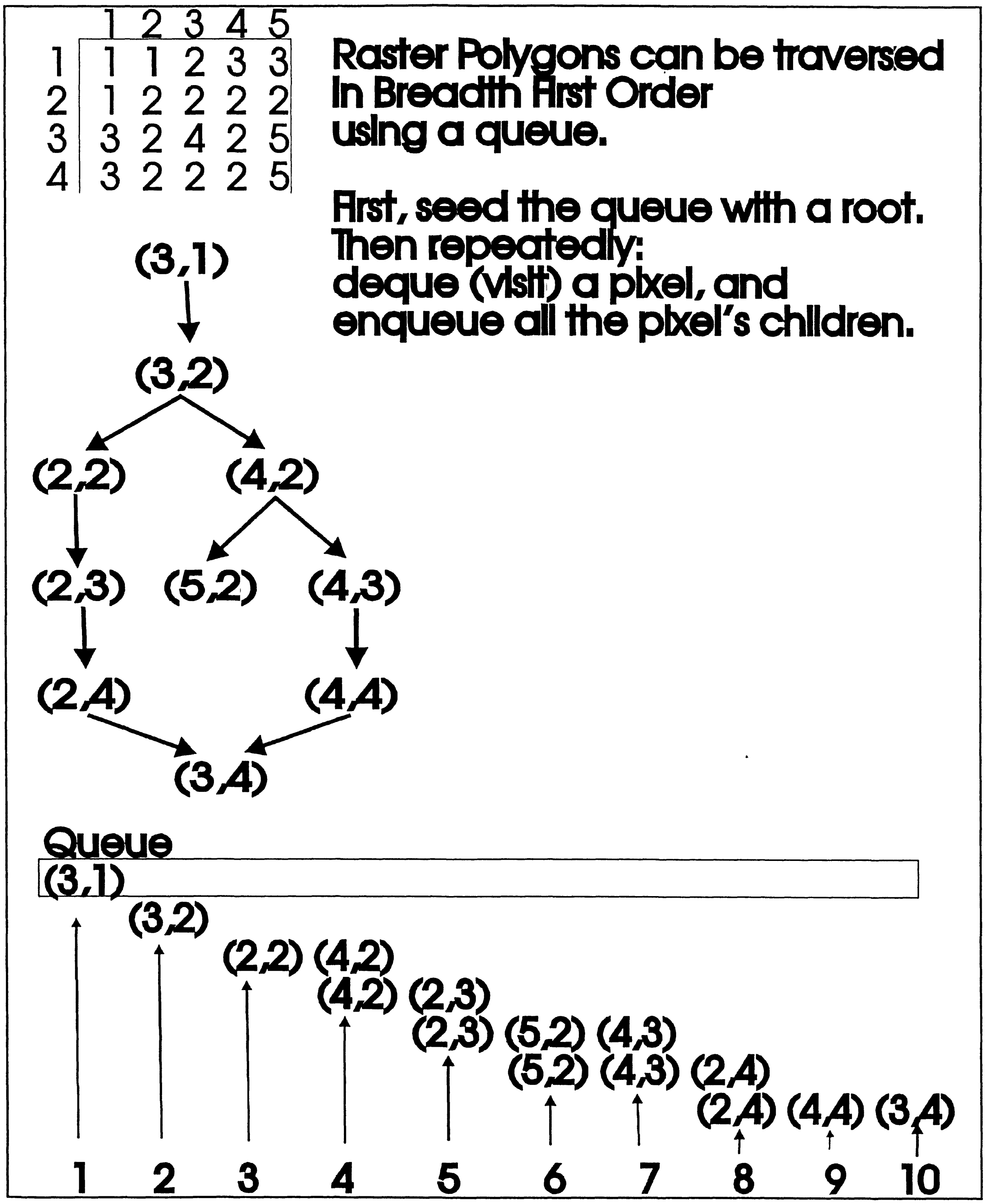

1 1 2 3 3
1 2 2 2 2
3 2 4 2 5
3 2 2 2 5
Images can be viewed as a collection of graphs
(1,1)
(1,2)
(2,1)
(3,1)
(3,2)
(2,2)
(4,2)
(2,3)
(5,2)
(4,3)
(2,4)
(4,4)
(3,4)
(4,1)
(5,1)
(1,3)
(1,4)
(3,3)
(5,3)
(5,4)

Figure 4.4



$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 1 & 1 & 2 & 3 & 3 \\
2 & 1 & 2 & 2 & 2 & 2 \\
3 & 3 & 2 & 4 & 2 & 5 \\
4 & 3 & 2 & 2 & 2 & 5
\end{array}
$$

Raster Polygons can be traversed In Breadth First Order using a queue.

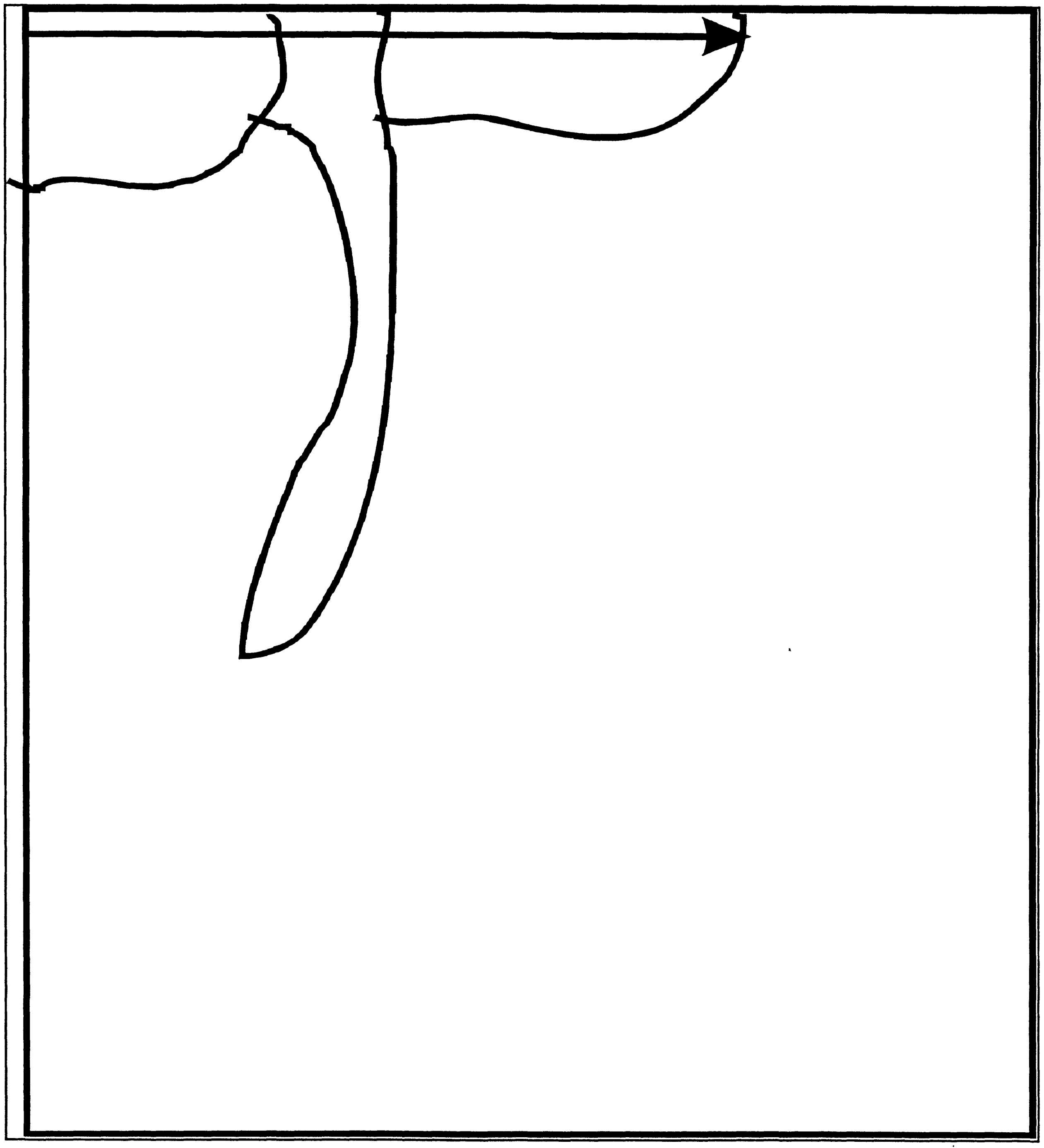
First, seed the queue with a root. Then repeatedly: deque (visit) a pixel, and enqueue all the pixel's children.

Using a queue to traverse the raster polygon formed by the image value 2

Figure 4.5

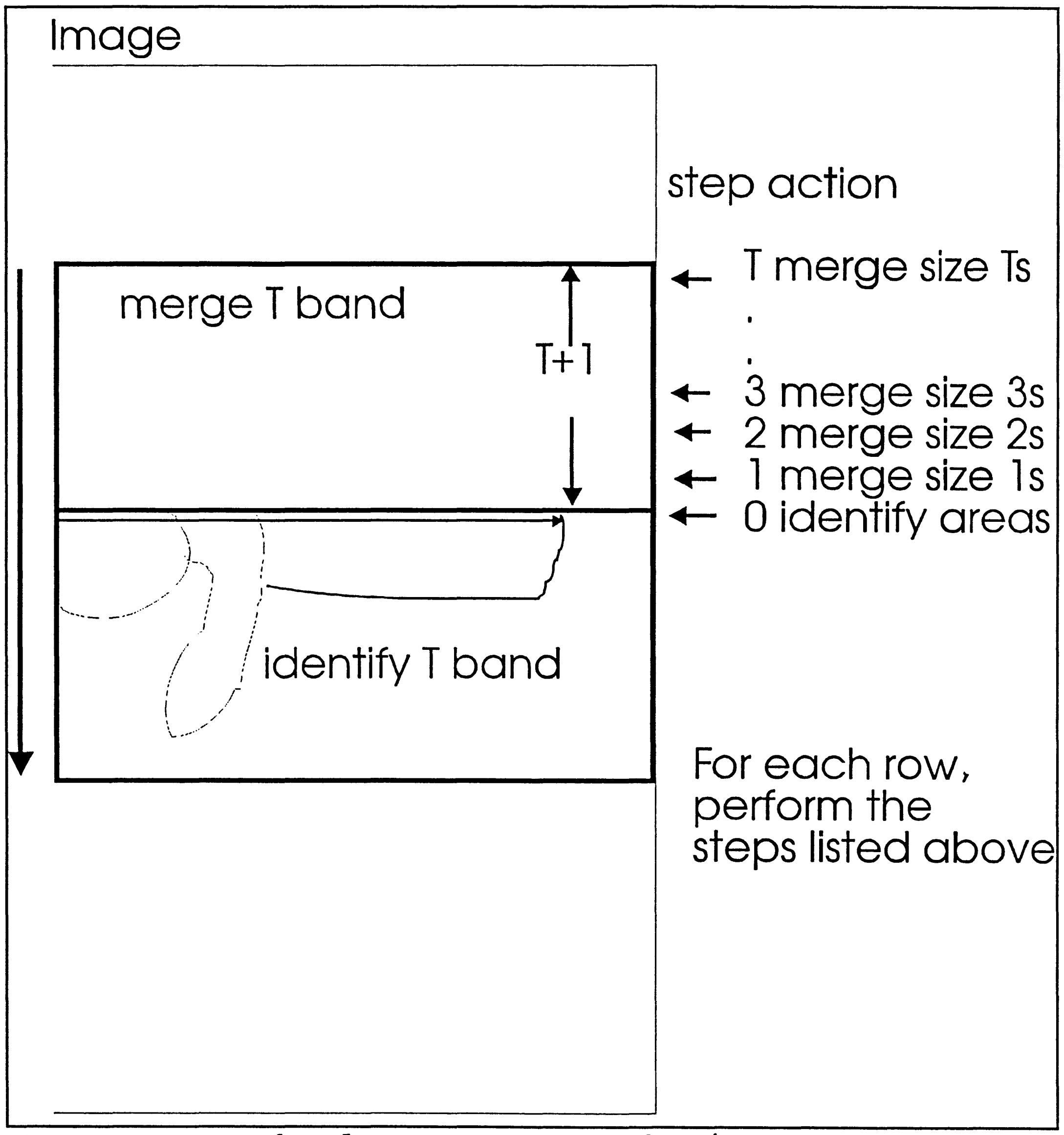
tBABA interleaves area identification and area merging. A moving memory resident band passes over the disk based image. At a central row in the band, TBMs are identified and placed in the **AreaContainer**. Then the merger immediately removes the size 1 areas and merges them, then one row back, size 2 areas are merged, then 2 rows back, size 3 areas merged, etc. until T-1 rows back, size T areas are merged. Then the band is advanced 1 row, and the process is repeated. This process is illustrated in Figure 4.7. In tBABA, when area identification sets bits in the last (T+1) row of a **BitBand** (corresponding the Identification Band), such areas are known to be a survivors. After the **Image** Band and **BitBand** are advanced, these survivors are extended by examining the last two rows in the Identification Band and setting the appropriate bits in the last row of the **BitBand**. In this way, survivors are marked identified and do not interfere with the identification of other areas.

In both designs, actual TN selection depends on the **SimilarityTable,** the **NoMergeTable**, and conceptually, the **TargetSelect** objects. The **SimilarityTable** is a function of two classified image values, which returns one real number that represents the 'similarity.' The **NoMergeTable** is simply a way to indicate that some class values should 'survive' no matter what area size, i.e., a table indexed by **Image** values pairs that returns a boolean value specifying whether or not to merge TBM with that value. ABA and RBR encapsulate the TN selection logic in the **TargetSelect** object.



ABA Area Identification. The image is scanned top to bottom, left to right. Upon encountering an unmarked area, it is traversed in breadth first order.

Figure 4.6



tBABA's Image band moves across the image one row at a time. At each row, area identification and subsequent, ordered merging are done.

Figure 4.7

5.0 Worst Case Analysis of ABA

5.1 Worst Case Analysis of ABA with Static Ordering (sABA), followed by a discussion of how this applies in practice.

Given an image X with R rows and C columns, we use I=R*C as the number of pixels, T as the threshold size, and A <= I as the number of raster polygons. For simplicity we assume that an image consists of 8-bit data values.

For a worst case space analysis we need to identify what objects use significant memory, and what objects or parts of objects must be resident at anytime. The space significant objects and worst case estimate of their size in bytes are shown in Table 5.1.

object	space required
Image	I
BitMap1	I/8
BitMap2	I/8
AreaContainer	6A
PointQueue	max(R,C)*4

Table 5.1

In sABA, all these objects are coresident, and the total space requirements is bounded by 5I/4 + 6A + 4max(R,C). Each TBM placed in the AreaContainer requires 6 bytes, 4 bytes for its area identifier and 2 bytes for its size. In a worse case each area is of size 1, so that A=I. To simplify the analysis, assume that X is square, so R=C=sqrt(I). Therefore, the worst case space requirement is: 29I/4 + 4sqrt(I). For large I, this is 7.5*I bytes. In our experience with real images, A is usually less than I/3. Thus, a useful rule to estimate the practical space requirement is 5I/4 + 6I/3 + 4sqrt(I), or approximately 3.5*I bytes.

The worst case time analysis is somewhat more difficult to estimate due to the dynamics of the process. However, we can consider the time to effect the aggregation as the sum of stages: area identification, ordering, TBM merges, and Image I/O. Below, we demonstrate that the worst case time complexity of these stages is O(I) + O(I log(I)) + O(TI) + O(I).

Part One: Worst Case Time for Area Identification.

The time to identify all the areas in an image is the total of time to scan the image, plus the time to traverse all the areas. The time to scan the image is simply I. The worst case is when every area is of size 1. The time to traverse an area of size S pixels is the time to read each pixel, enqueue and dequeue each point, examine its 4 neighboring pixels, and the time to mark the bit map as being traversed, respectively: 1S + 2S + 4S + 2S = 9I, which is O(S). Thus the total time to traverse all areas is O(I).

Part Two: Worst Case Time for Ordering.

The ordering stage is done by repeated heap insertions, known to be Nlog(N). In worst case N=A where A<=I. Therefore, the worst case ordering is O(A log(A)) <= O(I log(I)).

Part Three: Worst Case Time for TBM Merges.

As a proof that the worst case for the merge stages is O(TI), consider that each merger increases a TN's size by at least 1. Each survivor is the result of at most T mergers. Since there can be at most I/T survivors, it takes at most (I/T)*T, or I, mergers to bring all areas in an image to survivor size. For each merger, an individual pixel can be referenced at most T times. Therefore, the upper bound for the merge stage is O(TI).

Part Four: The input and output stages are just file I/O, which is O(I).

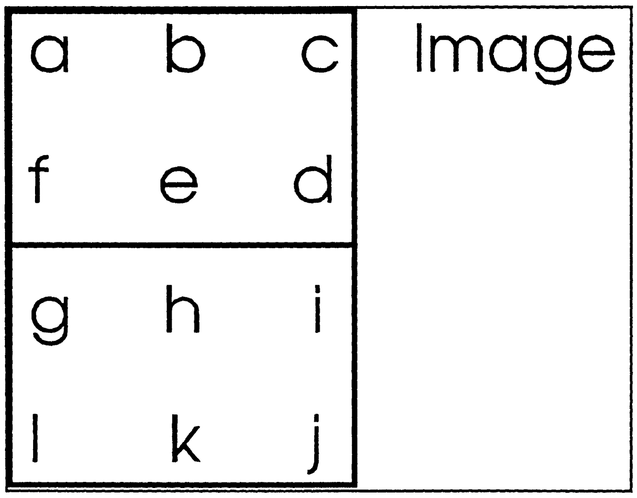
The example shown in Figure 5.2 reveals the pattern of worst case TBM merges. The image in the example is 3 columns wide and 4 rows deep, and the threshold is 6. The pixels are labeled a thru l. The resultant image contains I/T or 2 survivors. In this example, all TBMs are originally size 1. T mergers are needed to bring the original areas to survivor size T. The total merge time is the number of resultant survivors times the traversal time forming each survivor with a series of T TBM mergers. Using sABA the worst case to form a single survivor occurs when the TN of the current merge becomes the TBM in the next merge which, in our example, forces the sequence of traversals to occur that is illustrated in Figure 5.2. From this example, it follows that in the general case, an individual survivor is formed by

$2\sum_{i=1}^{T} i=T(T+1)$ pixel references.

A similar traversal would be made for all I/T survivors. The total time to merge all the TBMs with sABA is I/T * T(T+1), or O(TI).

Therefore, the worst case time analysis of the entire aggregation is O(I log(I)) + O(TI).

In practice, we can approximate this as O(TI) for the following reasons. Even though possible values of T can be less than log(I), due to the constant costs in area traversal, the merge stage dominates the aggregation process, taking about 4 times longer. This reasoning is also supported by the benchmarking results reported below, which indicate that all implementations are very sensitive to threshold size, not just image size.



traverse TBM a to find TN b, traverse again to effect the merge;
traverse TBM a,b to find TN c, traverse again to merge;
traverse TBM a,b,c to find TN d, traverse again to merge;
traverse TBM a,b,c,d to find TN e, traverse again to merge;
traverse TBM a,b,c,d,e to find TN f, traverse again to merge;
traverse TBM a,b,c,d,e,f to form the survivor, traverse again to reset the Bitmap.

Figure 5.2

5.2 Worst Case Analysis of Area by Area with Dynamic Reordering (dABA), followed by a discussion of how this applies in practice.

Continuing with the sort of analysis introduced in Section 5.1, the worst case space requirement for dynamic reordering is the same as for static reordering, about 7.5*I bytes.

For a worst case time analysis, the time for area identification and ordering remain the same as sABA, O(I) + O(I log(I)). However, the time required for TBM mergers is different and more complex. The worst case is when TBM's, starting at size 1, are traversed and merged, then traversed again and reinserted into the area heap because, during subsequent encounters, their size has changed (this is the method used in dABA). This will continue with size 2 areas, with size 4 areas, etc. until the TBM becomes a survivor. Merging to area sizes other than 2^n results in faster, (not worst case) arrival at the survivor size. This must be done for all I/T survivors.

As a proof that for dABA the merge stage is O(I log(I)), consider that the time spent merging is the sum of time spent referencing pixels plus the time spent reinserting areas back into the binary heap. As with sABA, it takes at most I mergers to bring all size 1 areas to survivor size. With dABA every TBM is merged into a TN that is at least as large as itself. So, any given pixel can be referenced at most 3log(T) times. Therefore, the number of pixel references is O(I log(T)). The number of reinsertions can be no more than the number of mergers I. Each heap insertion is O(log(I)). Therefore, in worst case, the merge stage is

O(I log(T)) + O(I log(I)), or O(I log(I)); end of proof.

The example illustrated in Figure 5.3 reveals the worst case pattern of TBM mergers. A single survivor is formed by the following T-1 steps:

Step 1: traverse TBM a; traverse again to merge to b;
traverse TBM ab; reinsert in heap, traverse again to reset bitmap;

Step 2: traverse TBM c; traverse again to merge to d;
traverse TBM cd; reinsert in heap, traverse again to reset bitmap;

Step 3: traverse TBM e; traverse again to merge to f;
traverse TBM ef; reinsert in heap, traverse again to reset bitmap;

Step 4: traverse TBM g; traverse again to merge to h;
traverse TBM gh; reinsert in heap, traverse again to reset bitmap;

Step 5: traverse TBM ab; traverse again to merge to cd;
traverse TBM abcd; reinsert in heap, traverse again to reset bitmap;

Step 6: traverse TBM ef; traverse again to merge to gh;
traverse TBM efgh; reinsert in heap, traverse again to reset bitmap;

Step 7: traverse TBM abcd; traverse again to merge to efgh;
traverse TBM abcedfgh; recognize it as survivor; traverse again to reset bitmap.

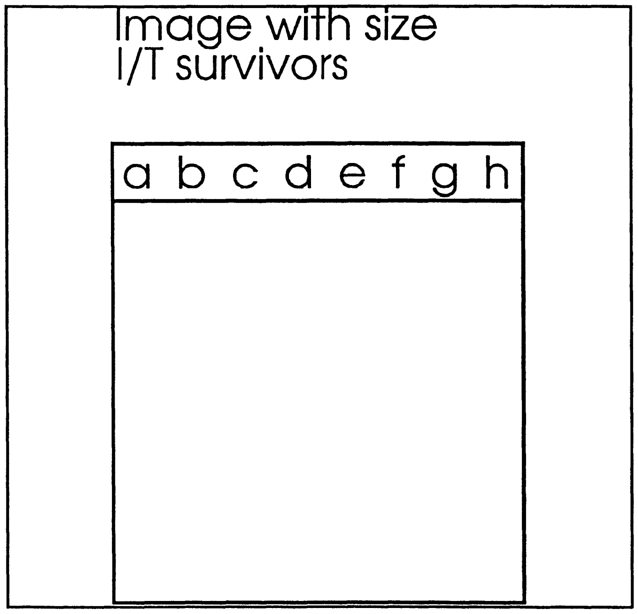


Figure 5.3

To reach survival size, the time spent on pixel references and reinsertions into the heap must be considered. In the example, the following summarizes the time spent merging as 'steps(pixel-references+reinsertion)':

Step 1-4:	4*(6+log(I))
Step 5-6:	2*(12+log(I))
Step 7 :	1*(24+log(I))

In the general case, the time needed to merge T size 1 areas to survival size is

$$\sum_{i=0}^{\log(T)-1} 2^{i}(3(2^{\log(T)-i})+\log(I)),$$

which simplifies to

$$3T\log(T)+T\log(I)-\log(I).$$

When multiplied by I/T survivors, the merge takes approximately

$$3I\log(T)+I\log(I)$$

The merge with dynamic reordering is therefore O(Ilog(I)).

In practice, on average dABA performs slower for real images, because the worst case for sABA is contrived and not likely to occur, whereas dABA's worst case reordering is closer to what happens with real images.

5.3 Worst Case Analysis of Sliding ImageBand (tBABA)

The worst case space requirements for tBABA are greatly reduced compared to dABA and sABA. The significant objects and their worst case space requirement are shown in Figure 5.4.

ImageBand	C(2T+2)
BitBands	C(2T+2)/8
AreaContainer	4CT (only Points are stored in T heaps)

Figure 5.4

The total space requirement is approximately 6*CT bytes.

The worst case time analysis follows dABA closely. Minor differences are the time to order I areas into T heaps, which is O(I log(C)), and the time to shift 2(T+1) bands R times, which is O(TR). The total time required by tBABA is the sum of the time for area-identification, initial ordering, band shifting, TBM mergers, and reinsertions, or respectively

$$O(I) + O(I \log(C)) + O(TR) + O(I \log(T)) + O(I \log(C)),$$

or, assuming that TR<=I, O(I log(max(T,C))).

tBABA uses the same dynamic reordering as dABA. Therefore, run times are expected to be comperable to dABA. Performance test for small MMUs confirm that tBABA runs only slightly slower that dABA.

6 Performance and Comparison Tests

6.1 Platform and Test Images

Performance and comparison tests were run an a IBM RS6000 43p running at a clock speed of 133Mhz, with a 192 megabytes of physical memory, and 512 megabytes of virtual memory. The operating system was AIX 4.1.

Test images were chosen from a collection of Landsat Thematic Mapper images used in vegetation cover analysis by the Wildlife Spacial Analysis Laboratory. Specific images were selected to be 'complex' in terms of both the number of original areas and the number of final areas. Test image names are p41r27.gis, p41r29u.gis, and p33r28u.gis, containing the number of original areas as approximately 13 million, 17 million, and 16 million respectively. The dimensions of these images are (as rows x columns), 7500x7890, 7520x7900, 6770x7136. Another test image was constructed by combining three p33r28u.gis images, concatenated horizontally, to produce an image having approximately 48 million original areas and dimensions of 6770x21408. A vertical concatenation would not stress row oriented implementation such as RBR/Ma.

Statistics were taken using the implementation of RBR created by Ma [8], along with my own sABA, dABA, and tBABA implementations. tBABA has not yet been optimized for speed with large MMUs, so only partial results are reported for tBABA performance tests, i.e., comparisons with it were only made for T=22 (or MMU=23).

Performance statistics were gathered from the 'gmon' program which reports real execution time, machine execution time, space utilization, and the number page faults generated. Complete performance results are given in Appendices A, B, and C.

6.2 Performance Tests Summary

RBR/Ma, sABA, and dABA perform alike (about 15 minutes each, clock time) on 60 megabyte images at low MMU's, i.e. 1-23 where 1 is the MMU of the original image and 23 is the MMU of the resultant image. RBR/Ma produced runs that were a few minutes faster than ABA in real time, but a few minutes slower in machine time. At greater MMUs or wider images, the RBR/Ma implementation quickly runs out of memory, but sABA and dABA continue to run. In fact, the ABAs run a variety of MMU values, even on the test image 3 times larger than the real image. As image size increases, virtual memory thrashing is seen in ratio of real time vs. machine time, producing increased clock time penalties on machines with relatively small physical memory.

As idicated by the tables in Appendix A, performance times for RBR/Ma and sABA are approximately linear with the number of TBMs in the input image. dABA shows a noticeable slowdown at large threshold values, reflecting the sensitivity to 'T' discussed above. For example, at an MMU of 1112, dABA takes 5 times longer than sABA.

One technique that can facilitate processing for large MMUs is to pipeline several runs with increasing MMUs, i.e. perform aggregation processing of 1-1000 by a sequence of runs 1-23, 23-100, 100-200, ..., 900-1000. With complex images, RBR/Ma needs at least 3 runs to produce outputs with MMUs of 445. Though not analyzed here, RBR/Ma's performance is much more influenced by the number of TBM's for which descriptors must be maintained, and the ratio of band size to threshold size. For RBR/Ma, at each stage its run time increases while corresponding ABA run time decreases. ABA's run times were faster than RBR/Ma's by a factor of 8 for MMUs of 445. Pipelining several ABA runs produced combined run times similar to a single run to the same MMU.

Although detailed testing has yet to be performed, the current version of tBABA takes about 35 minutes on 60 megabyte images. It uses very little space. Although possible, tBABA has not yet been optimized for large MMUs as sABA and dABA have been.

6.3 Pixel Difference Comparisons

As the tables in Appendix B indicate, particularly with large MMUs, slight differences in merge order and cascade handling can produce very different final images. Images produced by the various implementations at the same MMUs were compared pixel by pixel. Pixel differences ranged between 20 percent with small MMUs, to 66 percent with large MMUs. In terms of pixel differences, RBR/Ma, sABA, dABA, and tBABA produced images almost equally different. RBR/Ma compared slightly better to dABA. tBABA compares best with dABA.

6.4 Pixel Similarity Comparisons

Processing an image with any one of the implementations transforms each pixel value in the original image into a corresponding pixel value in the resultant image. The similarity of corresponding pixels, as defined by the similarity function, can provide a measure of information lost from the original image during aggregation. A pixel value transformed to one with great similarity loses little information; a pixel value transformed to one of less similarity loses more information.

Two resultant images can be compared by categorizing the difference in information loss for each pixel via the similarity difference. Rather than simply aggregating numerical differences, we classify the differences in ranges to produce a frequency distribution of pixel differences per category. This is illustrated by the histogram in Figure 6.2. Although the categories chosen are somewhat arbitrary, this technique offers a potentially better means of comparison than numerical summation of differences, particularly in cases where similarity values are not uniformly spaced.

Pixel similarity comparisons are presented for MMUs 1-23 (the smallest area in the original image is size 1, and the smallest in the resultant image is size 23) in Appendix C to illustrate the approach. However, we have yet to apply results from ground truthing to this type of comparison, so it is premature to attempt to draw conclusions from this data.

Another possible approach to image analysis using similarity is to view the similarity function as a 'membership function' of a fuzzy set [9]. Fuzzy set theory is sufficiently formalized to provide many avenues of analysis that may be applicable to our work here.

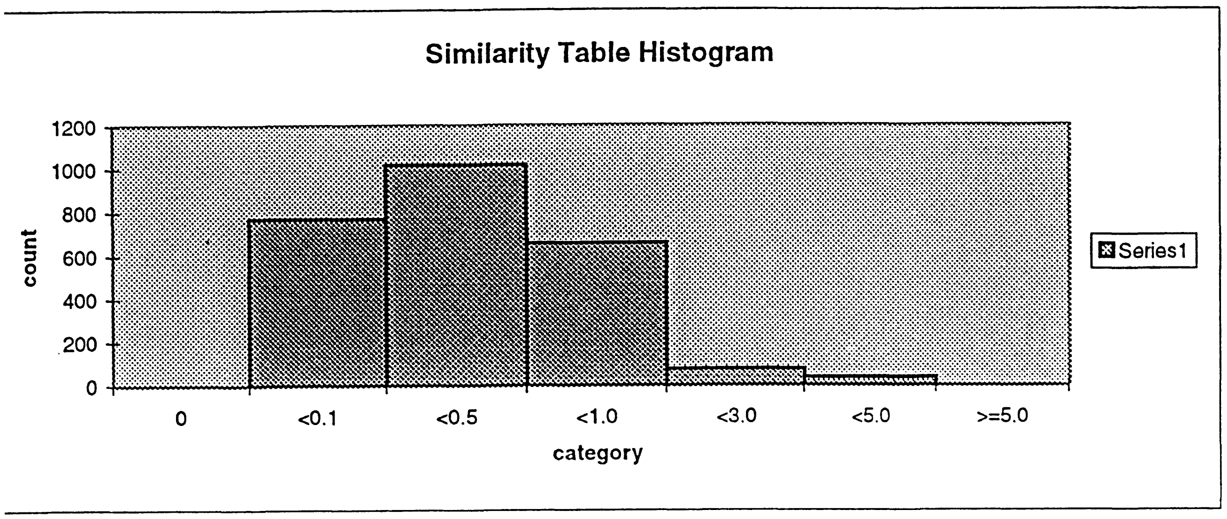


Figure 6.1

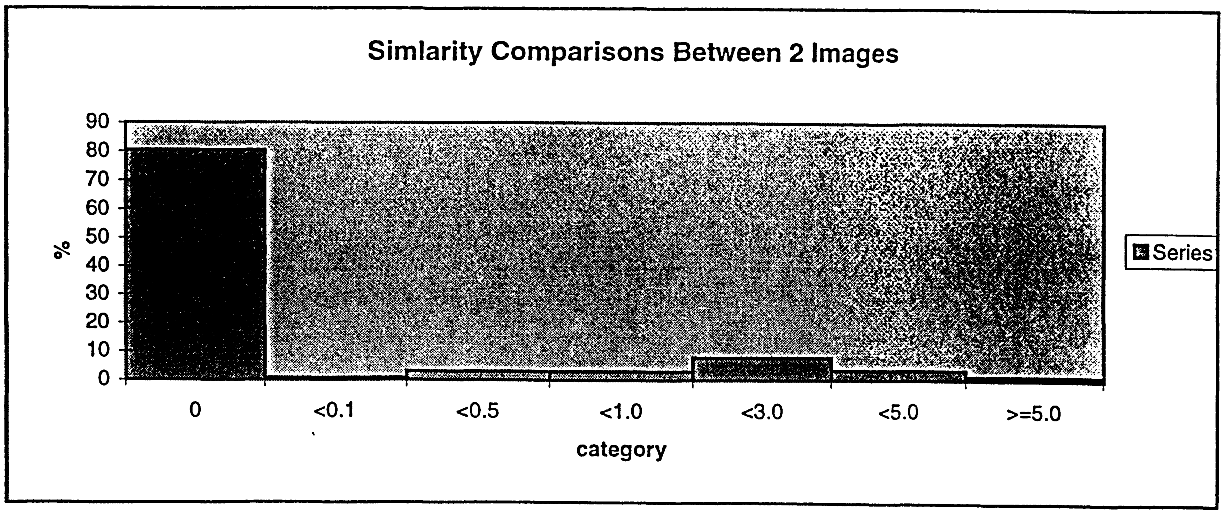


Figure 6.2

7 Conclusion

The objective of implementing ABA was to investigate alternatives to RBR designs which reduce space, and possibly time, requirements for large image aggregation. Both worst case space analysis and performance testing reveals that tBABA implementations can extend aggregation processing to 4 GigaByte images, even on simple workstations. This improvement is due to the reduced space required by the image band-processing, and the elimination of explicit **PointSets** and **NeighborLists**. At the time of this writing, tBABA has not been optimized for larger MMU's, so tBABA has only be tested for small threshold values. When optimized, tBABA is expected to run comparably to dABA with larger MMUs.

sABA and dABA extend the aggregation paradigm to images three times larger/wider than typical (row-width bounded) implementations of RBR can process. The breadth first traversal of areas provides area manipulation that is fast enough to produce run times as good as or better than RBR, without the overhead of dynamically manipulating lots of complex internal structures. The ABA approach extends easily to either static or dynamic ordering of the merge sequence. All ABA versions are also relatively easy to implement and do not maintain many complex internal structures, which tend to be more difficult to design and implement than direct image transformations.

With large MMU's and complex images, merge order has significant effects on the output image. All four implementations tested here produced final images that were different. It is easy to simply count pixel differences between two results, but more difficult to measure 'good' vs. 'bad' results. We have proposed the frequency distribution of similarity differences as a possible measure, but leave it to the application experts to assess its effectiveness.

8 References

[1] Ford, Ray; Ma, Zhenkui; Redmond, Roland. 1993. Aggregation of Image Classification Units for Mapping Large Areas, "Northwest Arc/Info Users Conference."

[2] Koltun, John. 1993. Classifications of Satellite Imagery in the Development of Wildlife Habitat Types, "Earth Observation Magazine."

[3] Copty, Nawal; Ranka, Sanjay; Fox, Geoffrey; Shankar, Ravi. 1994. Solving the Region Growing Problem on the Connection Marching, "NPAC Technical Report SCCS-397, Syracuse University."

[4] Berry, Michael; Comiskey, Jane; Minser, Karen. 1993. Parallel Cluster Analysis for Landscape Ecology Models, "Department of Computer Science, University of Tennessee."

[5] Ford, Ray. 1994. Raster Image Aggregation in Ecosystem Modeling, "Department of Computer Science, University of Montana."

[6] Guo, Jin. 1993. An Object-oriented model with efficient algorithms for Identifying and merging raster polygons, "Masters Thesis, Department of Computer Science, University of Montana."

[7] Ford, Ray. 1993. Automatic, Rule-Based Aggregation of Classified Remotely Sensed Imagery, "Department of Computer Science, Wildlife Spatial Analysis Laboratory, University of Montana."

[8] Ma, Zhenkui, 1996. Object & Rule Based Merging Module Version 4.06, "Wildlife Spatial Analysis Lab, The University of Montana."

[9] Gopal, Sucharila; Woodstock, Curtis; 1994. Theory and Methods for Accuracy Assessment of Thematic Maps Using Fuzzy Sets; "Photgrammetric Engineering & Remote Sensing, Vol. 60, No 2, Feb. 1994 p181-188."

[10] Lillesand, Thomas M; Keifer, Ralph W; 1979. "Remote Sensing and Image Interpretation."

Appendix A. Performance Test Results

Comparisons as reported by gmon between:
row-by-row area bound (RBR),
static ordering area-by-area (sABA),
dynamic ordering area-by-area (dABA), and
sliding threshold band area-by-area (tBABA).

MMU = 1-23
Real Time in seconds

	4127	4129	3328
RBR	973	1350	1180
sABA	1403	1667	1477
dABA	1800	2272	1999
tBABA	2449	2093	1717

MMU = 1-23
Machine Time in seconds

	4127	4129	3328
RBR	950	1112	1161
sABA	812	903	858
dABA	1214	1410	1350
tBABA	1486	1645	1546

MMU = 1-23
Space utilization

	4127	4129	3328
RBR	15744	15804	15704
sABA	176780	181856	177920
dABA	176628	181400	177476
tBABA	4484	4476	4088

MMU = 1-23
Page faults

	4127	1429	3328
RBR	54	20	2
sABA	26563	37414	30291
dABA	24180	38446	24656
tBABA	14496	14516	11807

Comparisons as reported by gmon between:

static ordering area-by-area (sABA), and
dynamic ordering area-by-area (dABA).

note: RBR requires more memory than is currently available.

MMU = 1-445
Real Time in seconds

	4127	4129	3328
sABA	2505	3083	2649
dABA	8116	9176	8834

MMU = 1-445
Machine Time in seconds

	4127	4129	3328
sABA	1888	2197	2004
dABA	7982	8249	7787

MMU = 1-445
Space utilization

	4127	4129	3328
sABA	177264	182656	180736
dABA	177812	178960	170780

MMU = 1-445
Page faults

	4127	1429	3328
sABA	28763	39208	32427
dABA	27503	41023	34896

Comparisons as reported by gmon between:

static ordering area-by-area (sABA), and
dynamic ordering area-by-area (dABA).

note: RBR requires more memory than is currently available.

MMU = 1-1112
Real Time in seconds

	4127	4129	3328
sABA	3627	4191	3706
dABA	17624	19356	17727

MMU = 1-1112
Machine Time in seconds

	4127	4129	3328
sABA	2946	3458	3121
dABA	16919	18303	17287

MMU = 1-1112
Space utilization

	4127	4129	3328
sABA	177204	179508	180684
dABA	177748	179564	179660

MMU = 1-1112
Page faults

	4127	1429	3328
sABA	31143	38791	33768
dABA	26622	41045	32022

Comparisons as reported by gmon between:
row-by-row area bound (RBR),
static ordering area-by-area (sABA), and
dynamic ordering area-by-area (dABA).

MMU = 23-223
Real Time in seconds

	4127	4129	3328
RBR	2691	8542	9031
sABA	1142	1290	1055
dABA	1326	1310	1175

MMU = 23-223
Machine Time in seconds

	4127	4129	3328
RBR	2646	8478	8970
sABA	636	656	625
dABA	775	808	787

MMU = 23-223
Space utilization

	4127	4129	3328
RBR	131228	131292	130316
sABA	77060	80776	64420
dABA	77496	81232	64868

MMU = 23-223
Page faults

	4127	1429	3328
RBR	124	158	69
sABA	18880	18987	15418
dABA	18903	19001	15341

Comparisons as reported by gmon between:
row-by-row area bound (RBR),
static ordering area-by-area (sABA), and
dynamic ordering area-by-area (dABA).

MMU = 23-445
Real Time in seconds

	4127	4129	3328
RBR	6590	6078	6075
sABA	867	993	693
dABA	897	860	675

MMU = 23-445
Machine Time in seconds

	4127	4129	3328
RBR	6481	5834	5575
sABA	353	357	307
dABA	357	362	314

MMU = 23-445
Space utilization

	4127	4129	3328
RBR	176324	176772	175844
sABA	73728	77148	60516
dABA	73740	77160	60532

MMU = 23-445
Page faults

	4127	1429	3328
RBR	14393	23021	22758
sABA	18559	18706	15255
dABA	18593	18682	15243

Large test image: three 33r28u.gis concatenated horizontally:

sABA with MMU of 1112

```
real time         : 23360 sec.
machine time      : 5447 sec.
memory utilization: 182276
page faults       : 838074
```

tBABA with MMU of 1-23

```
real time         : 5733 sec.
machine time      : 4724 sec.
memory utilization: 11364
page faults       : 35475
```

Appendix B. Pixel Difference Comparisons

Percent Pixel Comparisons:
RBR vs sABA vs dABA

p41r27.gis

MMU = 1-23	sABA	dABA
RBR	21.9	18.0
sABA		19.5

MMU = 1-445	sABA	dABA
RBR	*	*
sABA		47.6

MMU = 1-1112	sABA	dABA
RBR	*	*
sABA		51.4

p41r29u.gis

MMU = 1-23	sABA	dABA
RBR	27.3	23.1
sABA		24.8

MMU = 1-445	sABA	dABA
RBR	*	*
sABA		47.0

MMU = 1112	sABA	dABA
RBR	*	*
sABA		50.2

p33r28u.gis

MMU = 1-23	sABA	dABA
RBR	34.4	29.3
sABA		30.9

MMU = 1-445	sABA	dABA
RBR	*	*
sABA		62.0

MMU = 1-1112	sABA	dABA
RBR	*	*
sABA		66.0

* - not enough memory to run

Percent Pixel Comparisons:
RBR vs sABA vs dABA

p41r27.gis

MMU = 23-223	sABA	dABA
RBR	36.5	36.0
sABA		25.6

p41r29u.gis

MMU = 23-223	sABA	dABA
RBR	32.3	30.1
sABA		29.6

p33r28u.gis

MMU = 23-223	sABA	dABA
RBR	41.8	39.2
sABA		38.1

p41r27.gis

MMU = 223-445	sABA	dABA
RBR	41.2	40.9
sABA		27.7

p41r29u.gis

MMU = 223-445	sABA	dABA
RBR	33.4	31.7
sABA		31.0

p33r28u.gis

MMU = 223-445	sABA	dABA
RBR	43.5	41.3
sABA		40.3

Percent Pixel Difference Comparisons to tBABA at MMU = 23

p41r27.gis

RBR	18.1
sABA	19.7
dABA	6.5

p41r29u.gis

RBR	23.1
sABA	24.8
dABA	9.4

p33r28u.gis

RBR	29.4
sABA	31.0
dABA	12.3

Percent Pixel Difference Comparisons between
RBR 1-23 to dABA 23-445
and
RBR 1-23 to 23-223 to 233-445

p41r27	38.5
p41r29u	20.3
p33r28u	26.6

Appendix C. Pixel Similarity Comparisons

Tabular frequency distribution per category:

```
orig   = /eis2/ford/d.merge/d.p41r27/p41r27.gis
file1  = /eis2/sjb/p41r27.1-23.4w.img
file2  = /eis2/sjb/p41r27.1-23.5w.img
sim    = /eis2/ford/d.merge/d.p41r27/p41r27.smx
dimensions = 7890 columns by 7500 rows
total number  of pixels   = 59175000

category <=     0 47622521   80.477%
category <=   0.1   448182  0.75738% 3.8795%
category <=   0.5  1772467   2.9953% 15.343%
category <=     1  1773764   2.9975% 15.354%
category <=     3  4702293   7.9464% 40.704%
category <=     5  2037392    3.443% 17.636%
category <= 1e+07   818381    1.383% 7.084%



orig   = /eis2/ford/d.merge/d.p41r27/p41r27.gis
file1  = /eis2/sjb/p41r27.1-23.zw.img
file2  = /eis2/sjb/p41r27.1-23.5w.img
sim    = /eis2/ford/d.merge/d.p41r27/p41r27.smx
dimensions = 7890 columns by 7500 rows
total number  of pixels   = 59175000

category <=     0 48515989   81.987%
category <=   0.1   418518  0.70725% 3.9264%
category <=   0.5  1609456   2.7198% 15.099%
category <=     1  1685589   2.8485% 15.814%
category <=     3  4476412   7.5647% 41.997%
category <=     5  1830299    3.093% 17.171%
category <= 1e+07   638737   1.0794% 5.9925%



orig   = /eis2/ford/d.merge/d.p41r27/p41r27.gis
file1  = /eis2/sjb/p41r27.1-23.4w.img
file2  = /eis2/sjb/p41r27.1-23.zw.img
sim    = /eis2/ford/d.merge/d.p41r27/p41r27.smx
dimensions = 7890 columns by 7500 rows
total number  of pixels   = 59175000

category <=     0 46246207   78.152%
category <=   0.1   498389  0.84223% 3.8549%
category <=   0.5  1952929   3.3003% 15.105%
category <=     1  2042121    3.451% 15.795%
category <=     3  5392448   9.1127% 41.709%
category <=     5  2230459   3.7693% 17.252%
category <= 1e+07   812447    1.373% 6.284%
```

```
orig   = /eis2/ford/d.merge/d.p41r27/p41r27.gis
file1  = /eis2/sjb/p41r27.1-23.zw.img
file2  = /eis2/sjb/p41r27.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p41r27/p41r27.smx
dimensions = 7890 columns by 7500 rows
total number  of pixels   = 59175000

category <=      0 48486515   81.938%
category <=    0.1   421106  0.71163% 3.9398%
category <=    0.5  1622236   2.7414% 15.177%
category <=      1  1697604   2.8688% 15.883%
category <=      3  4482838   7.5756% 41.941%
category <=      5  1816971   3.0705% 16.999%
category <= 1e+07    647730   1.0946% 6.0601%
```

```
orig   = /eis2/ford/d.merge/d.p41r27/p41r27.gis
file1  = /eis2/sjb/p41r27.1-23.4w.img
file2  = /eis2/sjb/p41r27.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p41r27/p41r27.smx
dimensions = 7890 columns by 7500 rows
total number  of pixels   = 59175000

category <=      0 47516223   80.298%
category <=    0.1   454814  0.76859% 3.901%
category <=    0.5  1787864   3.0213% 15.335%
category <=      1  1791514   3.0275% 15.366%
category <=      3  4731197   7.9953% 40.581%
category <=      5  2050165   3.4646% 17.585%
category <= 1e+07    843223    1.425% 7.2325%
```

```
orig   = /eis2/ford/d.merge/d.p41r27/p41r27.gis
file1  = /eis2/sjb/p41r27.1-23.5w.img
file2  = /eis2/sjb/p41r27.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p41r27/p41r27.smx
dimensions = 7890 columns by 7500 rows
total number  of pixels   = 59175000

category <=      0 55326239   93.496%
category <=    0.1   180467  0.30497% 4.689%
category <=    0.5   678921   1.1473% 17.64%
category <=      1   636190   1.0751% 16.53%
category <=      3  1508823   2.5498% 39.203%
category <=      5   599452    1.013% 15.575%
category <= 1e+07    244908  0.41387% 6.3633%
```

```
orig   = /eis2/ford/d.merge/d.p41r29/p41r29u.gis
file1  = /eis3/sjb/p41r29u.1-23.4w.img
file2  = /eis3/sjb/p41r29u.1-23.5w.img
sim    = /eis2/ford/d.merge/d.p41r29/p41r29u.smx
dimensions = 7900 columns by 7520 rows
total number  of pixels   = 59408000

category <=     0 44680679    75.21%
category <=   0.1   143645  0.24179% 0.97536%
category <=   0.5   889588   1.4974% 6.0404%
category <=     1   931138   1.5674% 6.3225%
category <=     3  3249591     5.47% 22.065%
category <=     5  2323218   3.9106% 15.775%
category <= 1e+07  7190141   12.103% 48.822%



orig   = /eis2/ford/d.merge/d.p41r29/p41r29u.gis
file1  = /eis3/sjb/p41r29u.1-23.zw.img
file2  = /eis3/sjb/p41r29u.1-23.5w.img
sim    = /eis2/ford/d.merge/d.p41r29/p41r29u.smx
dimensions = 7900 columns by 7520 rows
total number  of pixels   = 59408000

category <=     0 45694676   76.917%
category <=   0.1   128640  0.21654% 0.93807%
category <=   0.5   773791   1.3025% 5.6426%
category <=     1   812998   1.3685% 5.9285%
category <=     3  2976817   5.0108% 21.707%
category <=     5  2273936   3.8277% 16.582%
category <= 1e+07  6747142   11.357% 49.201%



orig   = /eis2/ford/d.merge/d.p41r29/p41r29u.gis
file1  = /eis3/sjb/p41r29u.1-23.4w.img
file2  = /eis3/sjb/p41r29u.1-23.zw.img
sim    = /eis2/ford/d.merge/d.p41r29/p41r29u.smx
dimensions = 7900 columns by 7520 rows
total number  of pixels   = 59408000

category <=     0 43162636   72.655%
category <=   0.1   146992  0.24743% 0.90482%
category <=   0.5   907677   1.5279% 5.5873%
category <=     1   954689    1.607% 5.8767%
category <=     3  3496789   5.8861% 21.525%
category <=     5  2694996   4.5364% 16.589%
category <= 1e+07  8044221   13.541% 49.517%
```

```
orig   = /eis2/ford/d.merge/d.p41r29/p41r29u.gis
file1  = /eis3/sjb/p41r29u.1-23.zw.img
file2  = /eis3/sjb/p41r29u.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p41r29/p41r29u.smx
dimensions = 7900 columns by 7520 rows
total number  of pixels   = 59408000

category <=     0 45675845   76.885%
category <=   0.1   128922  0.21701% 0.93883%
category <=   0.5   773320   1.3017% 5.6315%
category <=     1   810252   1.3639% 5.9004%
category <=     3  2987275   5.0284% 21.754%
category <=     5  2289342   3.8536% 16.671%
category <= 1e+07  6743044    11.35% 49.104%



orig   = /eis2/ford/d.merge/d.p41r29/p41r29u.gis
file1  = /eis3/sjb/p41r29u.1-23.4w.img
file2  = /eis3/sjb/p41r29u.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p41r29/p41r29u.smx
dimensions = 7900 columns by 7520 rows
total number  of pixels   = 59408000

category <=     0 44638550   75.139%
category <=   0.1   143722  0.24192% 0.9731%
category <=   0.5   888850   1.4962% 6.0182%
category <=     1   933623   1.5715% 6.3213%
category <=     3  3259991   5.4875% 22.073%
category <=     5  2329500   3.9212% 15.772%
category <= 1e+07  7213764   12.143% 48.842%



orig   = /eis2/ford/d.merge/d.p41r29/p41r29u.gis
file1  = /eis3/sjb/p41r29u.1-23.5w.img
file2  = /eis3/sjb/p41r29u.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p41r29/p41r29u.smx
dimensions = 7900 columns by 7520 rows
total number  of pixels   = 59408000

category <=     0 53785892   90.536%
category <=   0.1    73318  0.12341% 1.3041%
category <=   0.5   410306  0.69066% 7.2981%
category <=     1   421300  0.70916% 7.4936%
category <=     3  1364795   2.2973% 24.276%
category <=     5   912971   1.5368% 16.239%
category <= 1e+07  2439418   4.1062% 43.39%
```

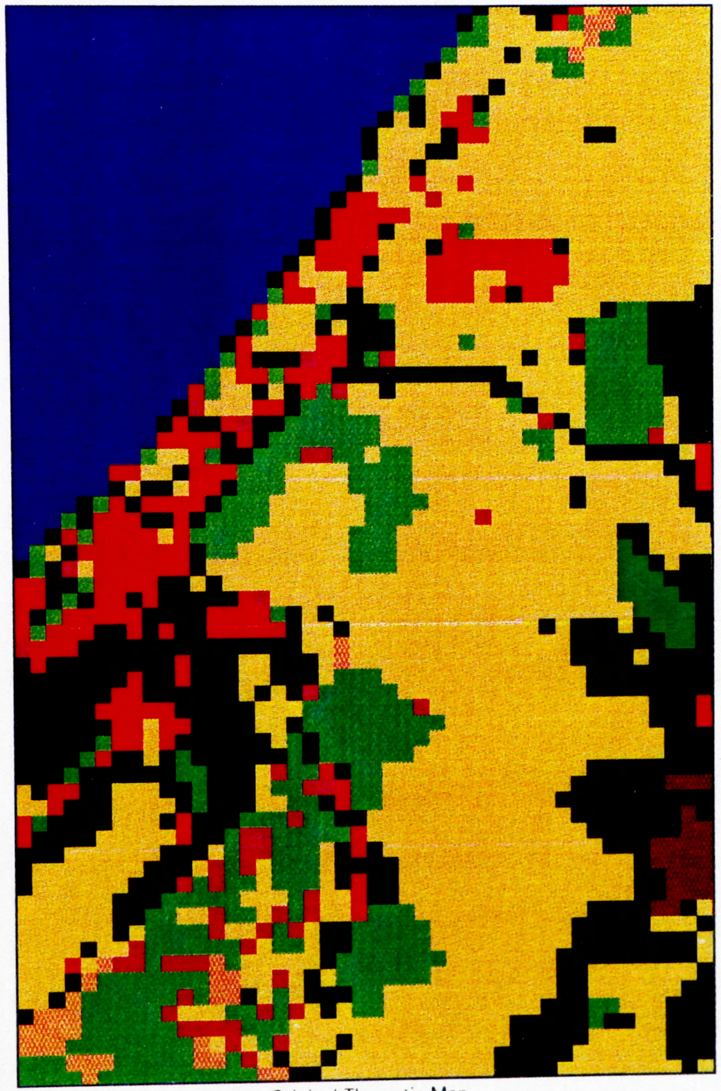
```
orig   = /eis2/ford/d.merge/d.p33r28/p33r28u.gis
file1  = /eis4/sjb/p33r28u.1-23.4w.img
file2  = /eis4/sjb/p33r28u.1-23.5w.img
sim    = /eis2/ford/d.merge/d.p33r28/p33r28u.smx
dimensions = 7136 columns by 6770 rows
total number  of pixels   = 48310720

category <=     0 33388271   69.112%
category <=   0.1   150730    0.312% 1.0101%
category <=   0.5   821530   1.7005% 5.5053%
category <=     1  1031804   2.1358% 6.9144%
category <=     3  3155459   6.5316% 21.146%
category <=     5  2296472   4.7535% 15.389%
category <= 1e+07  7466454   15.455% 50.035%
```

```
orig   = /eis2/ford/d.merge/d.p33r28/p33r28u.gis
file1  = /eis4/sjb/p33r28u.1-23.zw.img
file2  = /eis4/sjb/p33r28u.1-23.5w.img
sim    = /eis2/ford/d.merge/d.p33r28/p33r28u.smx
dimensions = 7136 columns by 6770 rows
total number  of pixels   = 48310720
total-sim/total-pixels    = 1.84735

category <=     0 34154504   70.698%
category <=   0.1   131018   0.2712% 0.92552%
category <=   0.5   722233    1.495% 5.1019%
category <=     1   932971   1.9312% 6.5905%
category <=     3  2948891    6.104% 20.831%
category <=     5  2293178   4.7467% 16.199%
category <= 1e+07  7127925   14.754% 50.352%
```

```
orig   = /eis2/ford/d.merge/d.p33r28/p33r28u.gis
file1  = /eis4/sjb/p33r28u.1-23.4w.img
file2  = /eis4/sjb/p33r28u.1-23.zw.img
sim    = /eis2/ford/d.merge/d.p33r28/p33r28u.smx
dimensions = 7136 columns by 6770 rows
total number  of pixels   = 48310720
total-sim/total-pixels    = 2.19649

category <=     0 31669777   65.554%
category <=   0.1   149265  0.30897% 0.89697%
category <=   0.5   828773   1.7155% 4.9803%
category <=     1  1074273   2.2237% 6.4556%
category <=     3  3465399   7.1731% 20.825%
category <=     5  2693333    5.575% 16.185%
category <= 1e+07  8429900   17.449% 50.658%
```

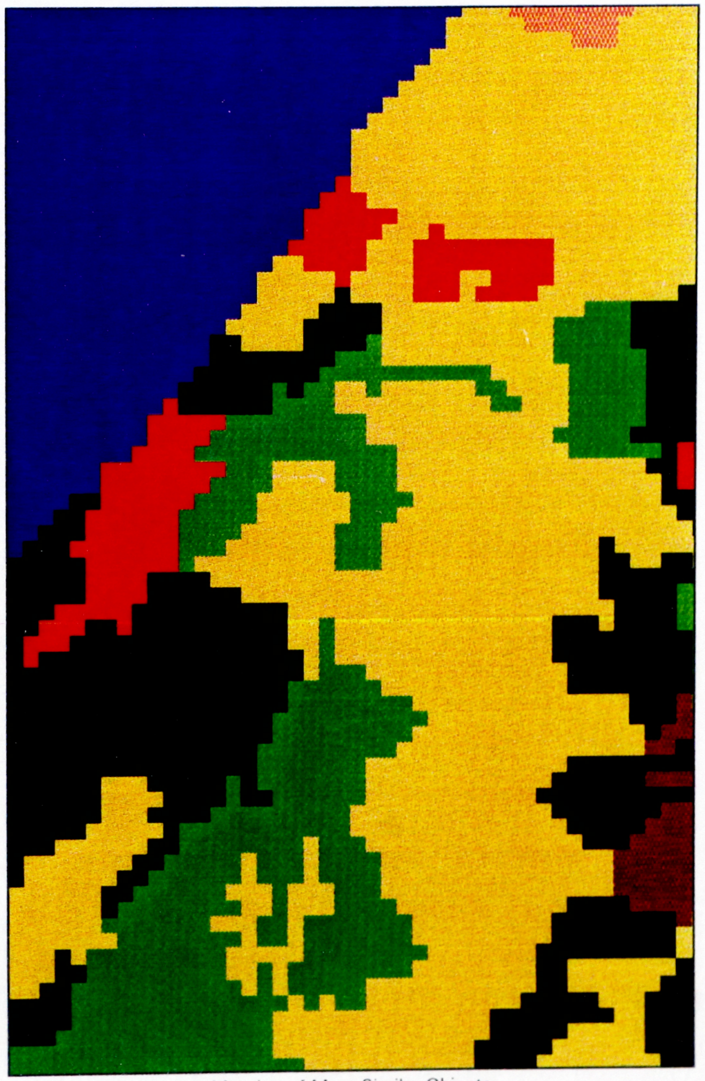
```
orig   = /eis2/ford/d.merge/d.p33r28/p33r28u.gis
file1  = /eis4/sjb/p33r28u.1-23.zw.img
file2  = /eis2/sjb/p33r28u.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p33r28/p33r28u.smx
dimensions = 7136 columns by 6770 rows
total number  of pixels   = 48310720
total-sim/total-pixels    = 1.84204

category <=     0 34117740   70.621%
category <=   0.1   131592  0.27239% 0.92716%
category <=   0.5   723607   1.4978% 5.0983%
category <=     1   938000   1.9416% 6.6089%
category <=     3  2977800   6.1638% 20.981%
category <=     5  2307234   4.7758% 16.256%
category <= 1e+07  7114747   14.727% 50.129%



orig   = /eis2/ford/d.merge/d.p33r28/p33r28u.gis
file1  = /eis4/sjb/p33r28u.1-23.4w.img
file2  = /eis2/sjb/p33r28u.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p33r28/p33r28u.smx
dimensions = 7136 columns by 6770 rows
total number  of pixels   = 48310720

category <=     0 33313116   68.956%
category <=   0.1   152857   0.3164% 1.0192%
category <=   0.5   827863   1.7136% 5.52%
category <=     1  1035801    2.144% 6.9064%
category <=     3  3168504   6.5586% 21.127%
category <=     5  2308940   4.7794% 15.395%
category <= 1e+07  7503639   15.532% 50.032%



orig   = /eis2/ford/d.merge/d.p33r28/p33r28u.gis
file1  = /eis4/sjb/p33r28u.1-23.5w.img
file2  = /eis2/sjb/p33r28u.1-23.82w.img
sim    = /eis2/ford/d.merge/d.p33r28/p33r28u.smx
dimensions = 7136 columns by 6770 rows
total number  of pixels   = 48310720

category <=     0 42368861   87.701%
category <=   0.1    79820  0.16522% 1.3434%
category <=   0.5   397958  0.82375% 6.6975%
category <=     1   483673   1.0012% 8.1401%
category <=     3  1394286   2.8861% 23.465%
category <=     5   944742   1.9556% 15.9%
category <= 1e+07  2641380   5.4675% 44.454%
```

A



Original Thematic Map

B



Merging of Most Similar Objects

[illegible]