1993

Object-oriented model with efficient algorithms for identifying and merging raster polygons

Jin Guo
The University of Montana

Let us know how access to this document benefits you.
Follow this and additional works at: https://scholarworks.umt.edu/etd

Recommended Citation
https://scholarworks.umt.edu/etd/5110

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

** Please check “Yes” or “No” and provide signature**

Yes, I grant permission  **Yes**
No, I do not grant permission  

Author’s Signature  

Date:  **Dec 17, 1993**

Any copying for commercial purposes or financial gain may be undertaken only with the author’s explicit consent.
An Object-Oriented Model with Efficient Algorithms for Identifying and Merging Raster Polygons

by
Jin Guo
Department of Computer Science, University of Montana

For the Degree of Master of Computer Science
University of Montana

Fall, 1993

Approved by

Chairman, Board of Examiners
Dean, Graduate School

12-28-93
Date
ABSTRACT

Raw images derived from satellite remote sensing systems are classified to produce a 2-D grid of
cells, which is maintained and manipulated in a cell-based geographic information system (GIS).
A grid is composed of raster polygons. Two important operations in a cell-based GIS are
identifying these raster polygons and eliminating small ones, which represent sampling "noise", by
aggregating them into larger polygons. Different applications may require different types of
identification and aggregation. Also, a given aggregation process may permit different
implementations, which can produce different result grids from the same input grid. We evaluate
several research efforts relating to this field and present a comprehensive analysis of the nature of
the aggregation problem. Formal specifications are proposed for operations ranging from polygon
identification to aggregation processes. Several criteria for comparing implementations of
aggregation processes are proposed. A set of rules for a specific application is analyzed. An object-
oriented design model is introduced as an additional discipline level between the specification and
algorithm design. Efficient algorithms for a specific rule implementation are presented, and
compared in terms of high level complexity analysis and performance results on a few test grids.

Key words: image processing, raster polygon, object oriented design, algorithm.
The research presented in this paper originates from a project implemented at the Montana Cooperative Wildlife Research Unit at the University of Montana. The focus on details of merge implementation stems from a sequence of implementations called "MMU" by Prof. Ray Ford. I contributed ideas in discussions for the continuing development of the MMU project, then started to implement new version of MMU by translating the Ada code into C++. Since I already had an Ada implementation at hand, I first chose to emulate the Ada program in C++. The work of emulation was very successful. The advantages of such emulation are that previous work on requirements and specifications can be preserved, and once the target emulation environment is properly set up, programming is more or less mechanical. Based on limited testing, the C++ emulation version appears to be about 20-30% faster than the original Ada version.

Like many software product developments, the development of the MMU project follows the spiral model of software development. The Ada implementation has evolved from version 1 (MERGE1) to version 18 (MERGE18) and the work is still going on. My C++ emulation version is based on MERGE9. By totally understanding the Ada program MERGE9, I found that significant improvements could be made in the algorithms. The specific goal of the MMU project was to be able to process image up to size of about 7500 x 7500, so memory use turned out to be a big issue. I decided to re-design the MMU implementation with this in mind, and reimplement it in C++. While Prof. Ray Ford continued to improve the Ada implementation, I designed an object-oriented model and made significant modifications the data structures and algorithms in the new C++ implementation. Compared with a newer MMU version MERGE11, efficiency both in time and space appears to be improved significantly; based on limited testing, the OOD C++ program appears to be about 5 times faster and uses about 20% less memory for a grid of 1000 x 1000 on the workstation IBM RS/6000-220.

For the OOD model design, I had to confirm some of the specifications which I felt were ambiguous. Because the given merge rules from the application requirements cannot strictly determine each merge step in the merge process, i.e., some freedom exists in the given merge rules,
different implementations produce different result grids for the same grid. Different interpretations of the same set of merge rules may greatly influence the complexity of the underlying algorithms. Thus I began to propose more formal specifications for the merge problem, along with criteria to measure "better" result grids so that different implementations and algorithms can be compared.

Acknowledgments

This paper is supervised and guided by my graduate committee, which consists of Prof. Ray Ford, Prof. Alden H. Wright and Prof. James R. Ullrich from the Department of Computer Science and Prof. Roland L. Redmond from the Department of Biological Science.

I am very grateful to Prof. Ray Ford for his introducing me to this research work and for his strict and patient directions as the committee chairman. We have shared a lot in the research. I discussed some of the theoretical issues with Prof. Alden Wright and he gave me some good ideas. As my academic advisor, he also gave me a lot of precious help during my whole graduate study.

Special thanks go to Prof. Roland Redmond and Dr. Zhenkui Ma. They offered very nice cooperation for this research work. I directly learned a lot of application background knowledge from Dr. Zhenkui Ma.
Abstract  ii

Preface  iii
  Acknowledgments  iv

1 Introduction  1

2 Related Work  6
  2.1 ARC/INFO and Its GRID Model  6
  2.2 The MMU Project  9
  2.3 Recent Work by M.Berry, J.Comiskey and K. Minser  12
  2.4 Recent Work by John Koltun  13
  2.5 Comments In General  15

3 Formal Specification of the Problem  16
  3.1 Basic Definitions  16
  3.2 Polygon Identification of Grid  21
  3.3 Merging Polygons on a Grid  23
  3.4 An Example of Rule Implementation  26

4 The Object Oriented Design Model  29
  4.1 The MIMRP Model and Its Main Classes  29
  4.2 Relationship Structure of the MIMRP Model  34

5 The Algorithms  36
  5.1 The Pseudocode, Classes and Operations  36
  5.2 The Identification Algorithm  43
  5.3 The Algorithm for the Merge Process  51
  5.4 The Algorithm for Building New Grid  52

6 Summary  55

7 References  56
1 Introduction

This research described here comes from an application project in the field of geoprocessing. Concepts, methods and techniques of image processing and geographic information system (GIS) are involved. Theoretical analysis and generalization in terms of computer science are applied to the implementation of the project. This chapter gives a simple and brief introduction to the background knowledge that is necessary to understand the context of this research work and its practical and theoretical significance.

Image processing, raw image and classified image

*Image processing* includes techniques like image reduction and magnification, contrast enhancement, edge detection and enhancement, and filtering, etc. These techniques are used for two somewhat different purposes [Russ-91]:

(a) improve the visual appearance of images to a human viewer and  
(b) prepare images for measurement of the features and structures present.

For visual enhancement, various methods like contrast enhancement and spatial filtering are developed by understanding the human visual process. The measurement of image features generally requires that the features be well defined, either by edges or unique (and uniform) brightness or color. Some results of image processing may be more appealing to the scenes, and others may convey more meaning.

*Raster digital images* (or *images*) come from many sources, but satellite imagery is the primary source used. A satellite image is similar to a photograph of the earth. The difference is that the image is made of bands, each band representing a spectral range of reflectance values of the features in the "photograph". Images store the spectral reflectance values of the features on the
ground; they do not identify the features themselves. Images that do not identify features are raw images or unclassified images. A raw image obtained from satellite is a 2-D grid of pixels. Each pixel represents a certain size square on earth. The features (or classes) can be identified through sophisticated classification algorithms. Some types of classification are implemented in commercial-available software packages like ERDAS, which also provide tools for other image-processing techniques. Other types of classification are being continually developed and improved by experts in a range of application domains.

One specific type of image classification attempts to identify classes of ground vegetation from analysis of satellite imagery. The general assumption in vegetation classification is: if the spectral reflectance in the image is directly correlated to vegetation conditions, then each pixel with the same spectral value should have the same vegetation conditions; furthermore, each pixel with a spectral value in the same spectral class of values should have very similar vegetation conditions. There are two possible ways to define spectral classes. In supervised classification, the user identifies areas of interest that define the class spectral statistics and parameters. In unsupervised classification, the user lets the classification program identify statistical clusters of spectral data that are significantly different. [Koltun-93, Jensen-86]

Through complex comparison of the spectral-reflectance values of different bands with an image-processing system, features such as vegetation classes can be identified, producing a classified image. Like the original image, the classified image is a 2-D grid of pixels, or cells. Each cell can be represented by a unique <ith row, jth column> in the grid. Each cell can contain a set of class values of that cell, including an identifier that defines to which class, group, or member the cell belongs, along with other data.

**What is and why GIS**

A geographic information system (GIS) can be described as an organized collection of computer hardware and software, geographic data, and personnel designed to efficiently capture, store, update, manipulate, analyze, and display all forms of geographically referenced information [ESRI#1-91]. The first geographic information system for storing and organizing spatial information in a computer was conceived about 27 years ago. Paralleling advancements in the GIS technology have been the growth of applications. From high quality cartography to land planning, natural resource management, environmental assessment and planning, tax mapping, ecological research, emergency vehicle dispatch, demographical research, and more, GIS promises to be one of the largest computer applications ever to emerge.
Chapter 1

There are two reasons for the rapidly growing interest in GIS. First, the costs of computer hardware needed for the tasks are dropping rapidly, and thus GIS processing is affordable to increasingly wider classes of users. More importantly, geography (and the data describing it) is part of our everyday world; almost every decision we make is constrained, influenced or dictated by some fact of geography. GIS technology helps us to organize data about the problems in the above mentioned applications and to understand their spatial relationships. Such an understanding is the basis for more sensitive and intelligent decision making.

Types of information in GIS

There are two basic types of map information [ESRI#1-91]:

1. spatial information describing the location and shape of geographic features and their spatial relationships to other features;
2. descriptive information about the features.

Information conveyed by a map is represented graphically as a set of map components. The locational information for components can be represented by points for features such as wells; lines for features such as roads; and areas for features such as lakes. More generally, map components can be represented in either vector form or raster form. For example, in a vector GIS system, a polygon is represented with a list of points corresponding to the vertices; while in a raster GIS, a polygon is represented by all the cells that it contains.

Descriptive information provides additional detail for the map components, like type and width of a road. In a typical GIS, such descriptive information is maintained in "attribute" relational databases. Spatial data and descriptive data can be linked by a common key, which is usually a unique identifier given to a spatial feature. This linkage allows a GIS to support queries that can answer many questions that an ordinary database system cannot answer.

Interface of GIS and image processing

With ordinary image processing tools, data can be derived about the study area that a raster GIS can not deduce. However, the information which the image processor can extract may be very useful as input to a raster GIS model. Such information can be used in the building of or addition to a GIS database. As a result, the two systems become synergetic, each providing additional
information and tools that the other system cannot. When used together, image-processing tools and raster GIS processing create a powerful analysis environment.

*Raster data*, as it relates to a raster GIS system, can be classified into three main groups: raster digital images from image processing system, raster GIS data from other GIS systems, and scanned imagery. With a raster GIS system like GRID in *ARC/INFO*, the raster data can be used for various processes: geometric transformation, edge matching for adjacent grids, boundary simplification and filtering small regions, just name a few.

**The application project**

This specific research described here is derived from an image processing requirement of the Montana Biodiversity Project in the Montana Cooperative Wildlife Research Unit at University of Montana. The Biodiversity Project requires a unique integration of satellite image processing and GIS analysis, beyond the "standard" capability implemented in commercial image processing or GIS packages.

The Biodiversity Project's vegetative classification problem starts with a seven band image. The three most sensitive bands for vegetative objects are chosen and the other four bands are discarded, thus condensing the original data. Since two instances of the same object may have slight variations, two pixels with minor reflectance value differences may actually represent two objects of the same type. The processing goal is to group pixels into *regions* that represent the presence of vegetation objects from different vegetation classes. The resulting classified image can be visualized by mapping the unique class identifiers to arbitrary colors, i.e., producing a false colored image. Additional details on the work of classifying raw images can be found in [Ma-93].

The classified image is a grid of cells, where each cell has only a single data value, the identifier of its vegetation class. The image can be refined by combining contiguous cells with the same class identifier into *regions*, called *raster polygons* (in contrast to *vector polygons*). For application purposes, either for visual enhancement or for analysis of specific features, regions whose size is below a *threshold* may be considered to be "noise" or "clutter", which must be eliminated by merging these small regions into their bigger neighbors. The details of this merge process are surprisingly complex, being subject to various ordering and neighbor-selection rules, such as by *similarity* or by length of common border. After the merge process, the new image will have no small regions smaller than the threshold size.

Thus, there are several aspects of the classification problem that follow the initial selection of class identifiers for each pixel. For a classified image, we must find efficient algorithms to identify
regions, to apply a set of merge rules to merge small regions into their bigger regions, and to generate the new image. Since there are various application specific requirements of merging and identification, the model for classification post-processing should be flexible and adaptable. In order to compare different merge algorithms, identification and optimal merging should be well defined. Also, because image maps are usually very large, a trade-off between memory usage and compute time can be a big issue in implementation. The goal of this thesis is to describe an approach to classification post-processing or what we call the "merge problem", which addresses these potential trouble spots.
2 Related Work

This chapter describes several research efforts related to the merge problem. Although the previous chapter provided some background knowledge, this chapter introduces details more closely related to the application and context of this research work. Discussions of each of the approaches and general comments given in Section 2.5 help explain how this research has combined and synthesized related work by others.

The terms region, cluster, area, polygon and raster polygon are used interchangeably here, as are image, map and grid, and pixel, point and cell. The meaning of a specific term should be clear from the context in which the term is used. Formal definitions of raster polygon and grid will be given in Chapter 3.

2.1 ARC/INFO and Its GRID Model

ARC/INFO is a powerful GIS composed of two parts: ARC and INFO. ARC deals with the spatial information and INFO deals with the descriptive information. The description of geographic information and features is exceedingly complex. A single data structure scheme simply cannot represent all geographic phenomena. Consequently, the ARC/INFO vector, grid, network and Triangular Irregular Network (TIN) data structures were each devised to produce highly specialized representations that work together to provide efficient access to a range of data types describing different spatial features. Accompanying these storage technologies are tools that have been developed to take advantage of the data structures. Combining the locational structures of grid, TIN and vector creates a powerful environment for geographic analysis. [ESRI#2-91]

GRID is a cell-based (raster) geoprocessing system integrated with ARC/INFO to permit processing of raster data structures. The model used in GRID is based on a combined 2-D grid and a relational attribute model. Pixel values in the 2-D grid provide keys into a relational database
system, which manages all actual attribute values associated with the grid. Data belonging to different themes are stored as separate grids, or "layers". Each categorical grid has an associated value attribute table (VAT) that is stored in an INFO relational database. Functional capabilities provided by GRID fall into four categories:

(a) those that work on single cells (local functions);
(b) those that work on cells within a neighborhood (focal function);
(c) those that work on cells with zones (zonal functions) and
(d) those that work on the whole grid.

In general, several storage and compression techniques for grids can be identified, such as cell-by-cell, run-length and quad tree. Cell-by-cell is for data that has a unique value for every cell, i.e., there is no way to compress the information. Run-length ([Jensen-86], pp. 47-48) coding stores data by row. When adjacent cells have the same values, "runs" are created to compress the data, as illustrated in Figure 2.1. The column range information is stored as a from-column to to-

column, along with the value associated with that run. Quad-tree coding first divides a grid (or matrix) into four equal, smaller squares when cells in the grid have different values. If all the four smaller squares have the same value, the square will not be broken down any further. Otherwise, the square that has different values has to be subdivided into four more equal squares. This process continues until each square only represents cells with the same values.

GRID compresses data using an adaptive run-length encoding scheme that automatically optimizes database size. It first tests each group of data to determine the optimal compression technique and then stores it in that manner. Spatial indexing and compression techniques are applied separately to each grid in a database. [ESRI#2-92]
Identifying and merging regions in *GRID*

1. Identifying regions: *GRID*'s `REGIONGROUP` function identifies contiguous regions, where "contiguous" is defined as an up, down, left, or right neighbor. It assigns an unique region identifier value to each region in a grid. The values are assigned by the scanning process, which starts in the upper-left corner of the grid and moves left-to-right, top-to-bottom. As each new region is encountered a unique value is assigned to it. This process continues until all regions are assigned a value. The output is a value attribute table (VAT) of the grid. Each record of the VAT consists of the class value for all the cells in the region, the size of the region and the original value of the zone the region belongs to. A zone consists of cells with the same value that are connected or disconnected or both.

2. Merging regions: There are two functions provided by *GRID* that can be used for merging regions. The `NIBBLE` function allows the user to replace the values for an area in a grid by assigning each cell in the area the value of the nearest cell on the boundary of the area. This area can range in size from a few cells to a region of many cells, and these regions do not need to be continuous. This is often useful for changing the values of regions where the present values are erroneous or unknown. This function uses `<in_grid>` and `<mask_grid>` as inputs. First, the program determines all areas in the `<mask_grid>` with the value `NODATA`. The corresponding areas in `<in_grid>` will be nibbled. The cells to be nibbled are changed incrementally from outside of the `NODATA` region, based upon the value of the nearest neighbor at the cell. This proximity is determined by Euclidean distance. The `MAJORITYFILTER` function can be used to remove small regions while maintaining many of the original spatial patterns. This function replaces the values of the cells based upon the majority of their continuous neighbors. The decision to replace the value of a cell is based upon two criteria, both of which can be controlled by the user. First, there must be enough neighboring cells of a given value, either a majority or half, depending upon the option chosen. Second, if there are enough cells, they must be continuous about the filter kernel. This second criterion concerning spatial connectivity for the cells minimizes the corruption of cellular spatial pattern. [ESRI#2-91]
Discussion

Although *ARC/INFO* is a flexible implementation of general GIS models, it may be inefficient or improper for some specific GIS applications. For example, the grid oriented data structures are very suitable for all the four kinds of functions that are supported. However, for the region merge process required by the Montana Biodiversity Project (see Section 2.2 for more details), a region oriented data structure is more efficient.

Another constraint is that data from region identification is not complete. Information such as the list of all cells in a region and neighbors of each region is not reported to the user. The merge operations supported do not meet merge requirements for applications such as the Biodiversity Project and the application described in [Koltun-93]. In these cases, similarity between classes must be considered.

2.2 The MMU Project

Introduction

For the Montana Biodiversity Project, small regions in the classified image need to be merged according to similarity between vegetative classes. Since the merge functions provided by GRID in *ARC/INFO* couldn't meet the application requirements, a project called *Minimum Mapping Unit (MMU)* was initiated by Ray Ford at the Computer Science Department of the University of Montana [Ford-93]. In the MMU project, all regions in the classified image are identified and small ones are merged into their neighbors according to the given merge rules. The MMU project has focused on the successful implementation of flexible merge process, and the successive refinement of the implementation into increasingly more efficient versions able to process larger and larger input data sets.

The inputs of the program are: a file of 2-D grid of cells each having an integer class number, a threshold size, and, for similarity-based neighbor selection, a class similarity array given in a separate file. The outputs are the new classified image file and a value attribute table for resulting regions.

Rules governing the merge process can be separated into ordering and neighbor selection. In general, the ordering rule is that all the to-be-merged polygons are sorted by size, in increasing order. When sorting, if two polygons have the same size, they are sorted by their positions (for
example, sorting by left-most point first and then by top point). Merging thus starts with the smallest polygon and proceeds along the sorted order.

There are two types of neighbor selection rules: similarity-based and application-process-specific. Similarity-based selection is focused on here. A polygon is merged into one of its neighbors, with which the merged polygon has the maximum similarity value. When the size of the resulting polygon is still smaller than the threshold this "surviving" polygon remains in its original place on list of to-be-merged polygons. The merging process continues until the to-be-merged polygon list is empty. The user also has the option of excluding regions with certain class values from the merge process, so that regions with this value will never be merged regardless of their size.

The polygon data structure is the most important data structure in the program. Each polygon has basic fields of class number, size, structure of neighbors and structure of points. The neighbor structure is a list of references to neighboring polygons. The point structure records locational information of all the points for the polygon. Figure 2.2 illustrates a polygon whose points range from column 1 (left-most column) to column 4 (right-most column). This data structure is similar to the run-length representation in GRID (see Figure 2.1). Note that when two polygons are merged, both the neighbor structure and the point structure must be adjusted for the surviving polygon. In addition, other polygons may have neighbor references to the "merged-away" polygon, which has logically ceased to exist.

![Figure 2.2 A raster polygon and its representation](image)

Points in a column for one polygon are thus represented as a list of integer pairs. The data structure for the list can be implemented as a linked list either with each element storing only one integer pair, or with each element containing an array of fixed size and thus storing more than one integer pair. The latter is adopted in the implementation of the MMU project.
The basic MMU algorithm

Identification:
Before the merge operation is applied, all polygons are identified. A two-row scanning algorithm for identifying polygons is implemented. Cells in a grid are processed in the order of left-to-right, top-to-bottom. Rather than reading the entire grid and storing it in memory at once, rows are read one by one into a two row buffer. Thus, when the processing of row $i$ is completed, row $i+1$ is read into the buffer, overwriting row $i-1$. When the whole grid has been scanned, all polygons are identified. The size, class value, locations of all the points, and neighbor information are recorded. In earlier MMU implementations, all neighbors of each polygon are recorded and the polygons are linked without sorting, then sorted in a separate pass. In later versions, neighbor information is recorded only for the to-be-merged polygons, and the polygons are sorted "on the fly" as they are identified.

Merging:
When polygons are identified and sorted, the merge algorithm designed for the given merge rules is applied to the to-be-merged polygons. Earlier versions of MMU implement the merge operation with two separate passes. First, each small polygon is merged into one of its bigger neighbors. Then, a second pass determines if merges have produced a cascaded effect, in which neighboring regions now have the same class value; if so, these regions are merged. In later versions of MMU, the check for cascaded effects is done at the time of each merge, eliminating the need for a separate pass.

Generating outputs:
Following all merges, a list of "active" polygons describes all regions in the grid, but the grid itself must be constructed. To generate the new image file, an array whose size is the same as the size of the original grid is declared. Each polygon writes all its points to the array. When all the polygons have been written out the array is output to the new image file. A value attribute table is generated. Each of the resulting polygons has a record in this table. The record has fields of class number, size, how many original polygons contained and the "Orig/End" ratio. The ratio gives a worst-case measure of "area accuracy", defined for each polygon as the percentage of the same class values in both the final and original images.
Discussion

(1) Compared with other identifying algorithms, the two-row scanning identifying algorithm has the advantage of processing big images with lower space complexity. The space complexity is only related to the number of polygons in grid, rather than to the $N^2$ number of pixels in a $N \times N$ grid.

(2) The point data structure for a polygon is like a run-length representation by column. This has advantages for the identifying and merge algorithms, but becomes a disadvantages when a big array must be used to generate the new output grid. Thus the benefit gained from the two-row scanning algorithm is compromised. The space complexity of the whole set of algorithms depends on both the number of polygons and the image size. A run-length representation by row for the point data structure can be designed to eliminate the limitation.

(3) Since the maximum image to be processed is about $7500 \times 7500$ with each cell taking a byte, memory use is a big issue. The space efficiency of the data structure for the list of integer pairs greatly depends on the related parameters like the size for the element array. A theoretical analysis on how to choose optimal parameter values needs to be done. Such an analysis is important and can be generalized to several other similar data structures like, the neighbor structure.

2.3 Recent Work by M. Berry, J. Comiskey and K. Minser

Introduction

Berry, et al., [Berry-93] describe work at the Department of Computer Science, University of Tennessee, using computer modeling to assess habitat fragmentation and its ecological implications. Maps (2-D grids) of habitat clusters are analyzed to determine number, sizes, and geometry of clusters. A scalable map analysis algorithm is produced for the identification and characterization of clusters for relatively large maps on massively parallel computers.

The existing sequential non-recursive Fortran-77 implementation of such models [Gardner-91] has limited the sizes and densities of clusters to be analyzed. To solve this problem, Berry, et al., have developed more efficient approaches that can exploit recursion or parallelism. They have implemented both recursive and non-recursive sequential C implementations, which achieved
significant improvements over the original code. They have also implemented parallel versions to get even more significant improvements.

Cluster identification in this paper is defined as: identify all clusters of 1's from an input map of 0's and 1's. This essentially involves locating and labeling clusters in a 2-D grid, and determine the total number of cluster, the size of each cluster, the largest cluster and the average cluster size. Adjacent pixels are considered to belong to the same cluster if they have the same value, and are up, down, left or right neighbors.

Cluster identification and labeling are accomplished by traversing the grid one cell at a time. When a cell containing a matching value is found, it is labeled with the current cluster value, and all subsequent nearest neighbors are also traversed. This process continues until all cells in a particular cluster are identified and labeled. Traversal of the grid then continues with the next unlabeled cell.

Discussion

(1) The largest processed image described in [Berry-93] is 1024 x 1024. Although the researchers claim that the limited size of maps and densities of clusters imposed in the original Fortran-77 program were improved, no analysis of space complexity was given.

(2) Each time only one map class can be analyzed. The grid element is either 1 or 0 where 1 represents a particular map class. If the map has more than one class, each map class must be analyzed individually by the algorithm.

(3) By the given meaning of cluster identification, the position of each pixel in a cluster is not recorded. Thus the identification is not complete.

(5) The map must be entirely stored in the memory before processing.

(6) There is no analysis of the communication cost between processors in their multiprocessor solutions, to see if the parallel solutions will scale to even larger images.

2.4 Recent Work by John Koltun

Introduction

Kultun [Koltun-93] reports on a pilot habitat mapping project implemented by the Geographic Resource Solutions (GRS), located at Arcata, California. The goal of this project was to generate
wildlife habitat types based on vegetation characteristics as defined by the *Wildlife Habitat Relations* (WHR) system. An objective and repeatable methodology was defined to generate detailed vegetation maps from satellite imagery and a unique integration of satellite image processing and GIS analysis in the classification of vegetation was involved. The researchers in the project found that the critical process between image classification and *polygon formation* for GIS has not been well defined. In order for the integration of image classification and GIS to be complete, an objective methodology was developed, including options for polygon creation from classified image data.

*Filtering* of various types is used to eliminate small polygons in classified image: *modal filtering*, *diversity filtering* and *linear preserving filtering*. However, researchers in the project found that these methods were useful for image enhancement, but not for polygon formation. The proper ecological choices for association between pixels were not considered.

A filtering method that would create a more realistic result would be based on the biological similarity of the pixels' vegetation attributes. The to-be-filtered pixel is assigned to the most similar type adjacent. During the aggregation process, if a type is below a certain similarity threshold with respect to its adjacent types, it will not be merged. This prevents the forced merging of significantly different types. For non-critical or similar types, the polygons are aggregated until the desired minimum type size has been achieved.

**Discussion**

Many operations in this project are very similar to those in the Biodiversity Project introduced in the first chapter. Both concepts of similarity are almost the same. The GRS researchers put emphasis on image classification and the interface with the GIS. No analysis of time and space complexity is given for identifying and merging polygons. The filtering method is at pixel level instead of at polygon level. The pixels to be filtered are not necessarily in the small polygons derived from the procedure of identification.

* MAJORITY function of GRID most probably implements this method.
2.5 Comments In General

The above mentioned research efforts provided different approaches for identifying and merging raster polygons to implement different application requirements. What information should be derived from the identifying operation depends on specific requirements. Different decisions may greatly influence the complexity of the identifying algorithm. There are variety of sets of merge rules required by different applications. Even with the same set of merge rules, if the rules are not defined strictly, two merge algorithms implementing the same set of rules may not be comparable with respect to complexity. When some algorithms are claimed to be more efficient than others it is necessary to have formal specifications of the problem.

Thus this research gives a comprehensive theoretical analysis on the problem and the related work. The formal specifications for identifying and merging raster polygons are given Chapter 3. When the problem is well specified, Chapter 4 gives a relatively general object-oriented model and Chapter 5 gives a description of the efficient algorithms that have been developed to significantly improve the program "MERGE9", version 9 of the MMU project implementation.
3 Formal Specification of the Problem

This chapter presents a formal specification of the merge problem raised at the end of Chapter 1. Definitions are given to objects, attributes of objects, relationships between objects and operations on objects. Identifying polygons and merging polygons on a grid are defined formally at a higher level. The specifications given here are not limited to any particular application but can be adapted and applied in general. Optimal criteria to measure better implementation of an aggregation process are also introduced and discussed in detail.

In the first two chapters, terms such as map, grid and image were used interchangeably. These terms loosely refer to the same thing. In this chapter, a strict mathematical definition of 2-D grid (or grid) is given and it is exclusively used throughout specifying the problem. Similarly, raster polygon (or polygon) is formally defined to represent the group of terms polygon, region, cluster and area, and cell is defined to represent terms cell, pixel and point.

3.1 Basic Definitions

To facilitate theoretical analysis, formal mathematical definitions are given to objects like grid and polygon, to object attributes like size of a polygon, to relationships between objects like neighboring polygons, and to operations on objects like polygon dissolution. If $X$ is an object, an attribute of $X$ is represented as $\mathcal{A}[X]$. For example, a polygon $P$ may have many attributes: $\text{size}[P]$ represents the size of $P$, $\text{cell}\_\text{set}[P]$ represents all the cells in $P$, $\text{start}\_\text{cell}[P]$ represents the start cell of $P$, and $\text{neighbors}[P]$ represents all the neighbors of $P$. More formal specification of these attributes will be given in the next chapter.

2-D Grid (or Grid in short). A grid is a two dimensional array of values drawn from an element data type. Each element of the array is called a cell of the grid. A grid can be represented by
$M_{R \times C}$, where $R$ is the number of rows and $C$ is the number of columns. $R \times C$ is the size of this grid, or $R \times C = \text{size}[M]$. $m_{ij}$ is a cell of grid $M$ that is located at row $i$ and column $j$. In this and following chapters, unless explicitly stated we assume that grids are square, and $n = R = C$.

**Class Numbers.** A set of class numbers can be defined as $S \subseteq \mathbb{N}$, where $\mathbb{N}$ is the set of integers. Usually, a class number is not negative. $\text{class\_value}[m_{ij}]$ or simply $m_{ij}$ is the class number the cell $m_{ij}$ takes.

**Neighboring Cells.** Suppose $m_{ij}$ and $m_{kl}$ are two cells. If they are adjacent to each other by north, east, west or south direction (also called NEWS directions), each is the neighbor of the other. For example, $m_{(i-1),j}$, $m_{(i+1),j}$, $m_{i,(j-1)}$, and $m_{i,(j+1)}$ are neighbor cells of $m_{ij}$.

**Raster Polygon** (or **Polygon** in short). A polygon $P$ is a set containing all neighboring cells that have the same data value, i.e., a polygon $P$ in grid $G$ is defined as 

\[ \{ a \mid a \in P \text{ iff } a \in G \text{ and } (\forall b \in P, \text{class\_value}[a] = \text{class\_value}[b]) \text{ and } (\exists b \in P, a \text{ and } b \text{ are neighbors}) \}. \]

**Exclusive Class Numbers.** This is a subset of class numbers $S$. Cells and polygons are treated specially if they take an exclusive class number. For example, polygons with exclusive class numbers will not be merged.

**Neighboring Polygons.** If polygon $P$ contains cell $m_{pi,qj}$ that is a neighbor cell with cell $m_{qi,aj}$ that is in polygon $Q$, then polygons $P$ and $Q$ are neighboring polygons.

**Threshold.** Threshold is a positive integer number. It is used to select polygons to be processed. For example, only those polygons whose sizes are smaller than the threshold can be merged.

**Similarity of Classes** (or **Similarity** in short). Similarity is a function $f: S \times S \rightarrow R$, where $S$ is the class numbers and $R$ is the set of real numbers. Similarity between classes can be represented by a two-dimensional array of real numbers, or by an upper or lower triangular matrix if similarity is symmetric.
Polygon Dissolution/Merge (or Dissolution/Merge Operation in short). Polygon dissolution is an operation by which the cells of a source polygon are moved into its one or more neighbors, called the target polygons. More strictly, suppose polygon $P$ has neighbor polygons of $N_1, N_2, ..., N_k$. $P$ is divided into $k$ polygons $P_1, P_2, ..., P_k$, such that for all $i = 1,2,...,k$, $(N_i, P_i)$ are neighboring polygons. Now, by doing $\text{class\_value}[P_i] \leftarrow \text{class\_value}[N_i]$ for all $i = 1,2,...,k$, we dissolve $P$. Note $P_i$ can be an empty set $\emptyset$. If $k-1$ of the subsets are empty, i.e., all of $P$ is moved into a single one of its neighbors, the dissolution is referred to as a polygon merge (merge operation).

In ARC/INFO's GRID package of raster operations, the NIBBLE function implements polygon dissolution and REGIONGROUP implements polygon merge (see Section 2.1).

Dissolution/Merge Process. Another way to represent dissolution/merge operations is to use a function: $g(P, M) = M'$, where $M$ is a grid, $P$ is a raster polygon in $M$ and $M'$ is the new grid after $P$ is dissolved. A dissolution process is a recursive function that performs dissolution operations successively to a grid and a sequence of polygons, i.e.,

$$G(M_0, <P_0, P_1, ..., P_k>) = g(P_k, g(P_{k-1}, ..., g(P_0, M_0)...) = M_{k+1}$$

where $M_0$ is the original grid, $P_i$ is a polygon in grid $M_i$ and $M_{i+1} = g(P_i, M_i)$ for $i = 0,1,2,...,k$.

Similarly, a merge process is a function that recursively performs merge operations on a sequence of polygons. For simplicity, we will represent a dissolution/merge process $G(M, <P_1, P_2, ..., P_k>) = M'$ on an arbitrary sequence of polygons as $G_{M-M'}$.

Note that a dissolution/merge process assumes that the polygons have already been ordered into the sequence $<P_0, P_1, ..., P_k>$--the process does not define that order. Also note that the same function $g$ is used for each individual dissolution/merge step, but no details are given for the function $g$.

Cascaded Effect. Suppose $P$ has neighbors $N_1, N_2, ..., N_k$. If, following the dissolution of $P$ there are polygons $N_i$ and $N_j$ that are neighbors and have the same class value, then according to the polygon definition, $N_i$ and $N_j$ represent a single polygon. This is called a cascaded effect of polygon dissolution/merge.

Polygon Truth* Ratio (or PT-Ratio in short). Assume a dissolution/merge process $G(M, <R_1, R_2, ..., R_k>) = M'$, where $P'$ is a polygon of $M'$ composed of polygons $P_1, P_2$,  

* "Truth" is used in this context in a manner consistent with the notation of "ground truthing" in ecosystem modeling.
...and \( P_h \) of the original grid \( M \). The PT-Ratio of \( P' \) for the dissolution/merge process \( G_{M-M'} \) is defined as

\[
PT(P', G_{M-M'}) = \frac{\sum_{m \in M'} f(i, j)}{\text{size}[P']}, \quad \text{where } f(i, j) = \begin{cases} 1, & \text{if } \text{class\_value}[m] = \text{class\_value}[m'] \\ 0, & \text{otherwise} \end{cases}
\]

PT-Ratio measures the extent to which the data values in the region of polygon \( P' \) in the new grid \( M' \) are the same as the data values in the regions \( P_1, P_2, \ldots, P_h \) in the old grid \( M \).

**Alpha Grid Truth Ratio** (or AGT-Ratio in short). Assume a dissolution/merge process \( G(M, < R_1, R_2, \ldots, R_k>) = M' \), where \( M' \) consists of \( P_1, P_2, \ldots, P_k \). The AGT-Ratio for the dissolution/merge process \( G_{M-M'} \) is defined as

\[
\text{AGT}_\alpha(G_{M-M'}) = \frac{\sum_{i=1}^{k} PT(P_i, G_{M-M'})}{k}.
\]

AGT-Ratio measures the extent to which the new grid resulting from the dissolution/merge process is the same as the original grid.

**Beta Grid Truth Ratio** (or BGT-Ratio in short). Assume a dissolution/merge process \( G(M, < R_1, R_2, \ldots, R_k>) = M' \). The BGT-Ratio for the dissolution/merge process \( G_{M-M'} \) is defined as

\[
\text{BGT}_\beta(G_{M-M'}) = \frac{\sum_{m \in M} f(i, j)}{\text{size}[M]}, \quad \text{where } f(i, j) = \begin{cases} 1, & \text{if } \text{class\_value}[m] = \text{class\_value}[m'] \\ 0, & \text{otherwise} \end{cases}
\]

Like AGT-Ratio, BGT-ratio also measures the extent to which the grid resulting from a dissolution/merge process is the same as the original grid.

**Gamma Grid Truth Ratio** (or GGT-Ratio in short). Assume a dissolution/merge process \( G(M, < R_1, R_2, \ldots, R_k>) = M' \), and let \( C_M \) be the number of polygons in \( M \) and \( C_{M'} \) the number of polygons in \( M' \). The GGT-Ratio for the dissolution/merge process \( G_{M-M'} \) is defined as

\[
\text{GGT}_\gamma(G_{M-M'}) = \frac{C_{M'}}{C_M}.
\]

The GGT-Ratio is another measure of the extent to which the new grid resulting from a dissolution/merge process is the same as the original grid.
Grid Truth Ratio (or GT-Ratio in short) is loosely defined as any measure of the extent to which the grid resulting from a dissolution/merge process is the same as the original grid. In this paper, it is used to refer to Alpha Grid Truth Ratio, Beta Grid Truth Ratio or Gamma Grid Truth Ratio.

Compatibility of Two Types of GT-Ratio. Assume that \( R_1 \) and \( R_2 \) are two types of GT-Ratio, and \( G_1_{M-M} \) and \( G_2_{M-M} \) are any two dissolution/merge processes with the same input grid and the same sequence of polygons. We say that \( R_1 \) and \( R_2 \) are compatible if
\[
R_1(G_1_{M-M}) > R_2(G_1_{M-M}) \Rightarrow R_1(G_2_{M-M}) > R_2(G_2_{M-M})
\]
or
\[
R_1(G_1_{M-M}) \leq R_2(G_1_{M-M}) \Rightarrow R_1(G_2_{M-M}) \leq R_2(G_2_{M-M}).
\]
Otherwise, \( R_1 \) and \( R_2 \) are noncompatible or competitive.

Let \( T_a \), \( T_b \) and \( T_g \) represent AGT-Ratio, BGT-Ratio and GGT-Ratio respectively for a dissolution/merge process. According to their definitions, \( T_a \), \( T_b \) and \( T_g \) must all be in the range \((0, 1]\), but it is possible in some cases to provide tighter bounds. For example, if in the dissolution/merge process every polygon is exclusive and no actual dissolution operations occur or there are no polygons that satisfy the to-be-merged definition, then \( T_a = T_b = T_g = 1 \). Thus, the maximum value for each GT-Ratio is in fact 1. On the other hand, the minimum value for \( T_b \) is \( T_b = 1/n^2 \), where a class number has only one cell in the original grid and it is used to replace all the other cells in the new grid. Similar lower bounds can be computed for \( T_a \) and \( T_g \).

![Figure 3.1 Illustration of PT-Ratio and GT-Ratios](image)
Based on their definitions, $T_{\alpha}$ and $T_{\beta}$ are compatible, but $T_{\gamma}$ is competitive with $T_{\alpha}$ and $T_{\beta}$.

Figure 3.1 illustrates the computation of PT-Ratio and GT-Ratios, and their compatibility relationships. In Figure 3.1, let $G_{1M\to M1}$ be the merge process from (a) to (b) and $G_{2M\to M2}$ be the merge process from (a) to (c). In (b), $P_{13}$ is the result of merging $P_1$ into $P_3$, and $P_{24}$ is the result of merging $P_2$ into $P_4$. In (c), $P_1$ and $P_2$ remain unchanged, and $P_{34}$ is the result of merging $P_3$ into $P_4$. All of the PT-Ratio values and GT-Ratio values are listed below:

<table>
<thead>
<tr>
<th>$PT(P_{13}, G_{1M\to M1})$</th>
<th>$= 9/11 = 0.82$</th>
<th>$PT(P_{24}, G_{1M\to M1})$</th>
<th>$= 11/14 = 0.79$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PT(P_{24}, G_{2M\to M2})$</td>
<td>$= 11/20 = 0.55$</td>
<td>$PT(P_{1'}, G_{2M\to M2})$</td>
<td>$= PT(P_{2'}, G_{2M\to M2}) = 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_{\alpha}(G_{1M\to M1})$</th>
<th>$= (9/11 + 11/14)/2 = 0.802$</th>
<th>$T_{\alpha}(G_{2M\to M2})$</th>
<th>$= (1 + 1 + 11/20)/3 = 0.850$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\beta}(G_{1M\to M1})$</td>
<td>$= 20/25 = 0.800$</td>
<td>$T_{\beta}(G_{2M\to M2})$</td>
<td>$= 16/25 = 0.640$</td>
</tr>
<tr>
<td>$T_{\gamma}(G_{1M\to M1})$</td>
<td>$= 2/4 = 0.500$</td>
<td>$T_{\gamma}(G_{2M\to M2})$</td>
<td>$= 3/4 = 0.750$</td>
</tr>
</tbody>
</table>

Figure 3.2 Computation of PT-Ratio and GT-Ratios

This example suggests some simple conclusions, namely that $T_{\alpha}$ and $T_{\beta}$ are compatible while $T_{\beta}$ and $T_{\gamma}$ are competitive. If we use $T_{\beta}$ as the criterion, $G_{1M\to M1}$ is the better merge process. However, if $T_{\gamma}$ is used as the criterion, $G_{2M\to M2}$ is better than $G_{1M\to M1}$.

### 3.2 Polygon Identification of Grid

*Polygon identification for a grid* is an operation on a grid which can be represented as a function: $F: Grid \Rightarrow \text{ListOfPolygons}$. More precisely, polygon identification of grid consists three aspects: (1) what kinds of polygons need to be identified; (2) for each polygon to be identified, what attributes need to be recorded and (3) how to dynamically maintain attributes of each identified polygon. In order to identify the polygons for a grid $M$, $M$ must be scanned (i.e., read from external memory). For large grids, the scan is expensive, so an ideal identification process would scan the grid only once. Once the scan process has been done, all the following operations proceed using the internal representation of the grid. The goal is to combine the scanning and identification processes, and possibly other processes such as dissolution/merge and construction of the result grid. Although in most cases all polygons in a grid need to be identified when a grid is scanned, there are some cases where only some of the polygons need to be identified. For example, Berry, et al.
[Berry-93], describe an application where all grid cells have data values of either 1 or 0, and only polygons whose data value is 1 need to be identified.

**Polygon identification** can include the identification of several possible polygon attributes, such as $size[P]$, $class\_value[P]$, $start\_cell[P]$, $cell\_set[P]$ and $neighbors[P]$ for polygon $P$. Other geometric attributes of a polygon such as mean squared radius, perimeter and center of gravity may be useful for some applications. These attributes can be computed on demand from $cell\_set[P]$, or computed once and stored as explicit attributes. For a set of polygon's attributes $S$, if no member of $S$ can be derived from the other members of $S$, $S$ is a **minimal** set of attributes. On the other hand, if attribute $f$ in $S$ can be derived from other attributes of $S$, $f$ is **redundant**. If all the other attributes can be derived from a given set of polygon's attributes without referring to the original grid, this set of attributes is **complete**; it is **incomplete** otherwise. When a polygon has a complete set of attributes, we say this polygon is completely identified. For example, the polygon attribute set $\{class\_value[P], cell\_set[P]\}$ is both minimal and complete. If all polygons in a grid $G$ are completely identified, we say that polygon identification for grid $G$ is complete. Figure 3.3 shows a grid and the identification of all the polygons. The identification for each polygon is minimal but not complete, because the locational information of cells, $cell\_set[P]$, is not included.

<table>
<thead>
<tr>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>row 1 2 2 3 3 7</td>
</tr>
<tr>
<td>2 1 2 3 3 7</td>
</tr>
<tr>
<td>3 1 3 3 3 7</td>
</tr>
<tr>
<td>4 1 4 4 4 1</td>
</tr>
<tr>
<td>5 1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Polynoms</th>
<th>start_cell</th>
<th>class_value</th>
<th>size</th>
<th>neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>&lt;1 1&gt;</td>
<td>2</td>
<td>4</td>
<td>{#2, #4}</td>
</tr>
<tr>
<td>#2</td>
<td>&lt;1 3&gt;</td>
<td>3</td>
<td>6</td>
<td>{#1, #3, #4, #5}</td>
</tr>
<tr>
<td>#3</td>
<td>&lt;1 5&gt;</td>
<td>7</td>
<td>3</td>
<td>{#2, #4}</td>
</tr>
<tr>
<td>#4</td>
<td>&lt;2 1&gt;</td>
<td>1</td>
<td>9</td>
<td>{#1, #2, #3, #5}</td>
</tr>
<tr>
<td>#5</td>
<td>&lt;4 2&gt;</td>
<td>4</td>
<td>3</td>
<td>{#2, #4}</td>
</tr>
</tbody>
</table>

**Figure 3.3** A grid and its polygon identification

Although we can derive other attributes from a complete set and a minimal set is preferred to save space, a complete and minimal set is not always proper in practice. It may not be necessary to produce a complete identification for every polygon. On the other hand, it may be cheaper and more convenient to compute some redundant attributes for a polygon during the scan process than to compute them later from other attributes. For example, neighbor information is relatively expensive to compute; though it can be computed from the $cell\_set$ attribute, it may be more efficient to compute it once and retain it as an explicit attribute.
When a list of polygons is identified from scanning a grid, how to maintain the attributes for each identified polygon during the dissolution/merge process is another big issue. It may be more effective to maintain some redundant attributes which are used frequently than to recompute them every time they are required. Some attributes of a polygon may be needed throughout the whole merge process, and must be updated following each dissolution/merge operation. Other attributes may only be needed temporally; to save memory, these attributes can be discarded when they are no longer needed. For example, $\text{size}[P]$ is usually needed throughout the whole dissolution/merge process, and it must be updated each time $P$ absorbs one of its neighbors. In contrast, if $P$ becomes a stay polygon during the dissolution/merge process, the neighbor information $\text{neighbors}[P]$ is no longer needed and may be discarded. Statistical information on all the polygons of a grid may be needed: the total number of polygons, maximum and minimum polygons, average polygon size and the size variance, etc. Such data can be collected either at the time the grid is being scanned or at later computation [Berry-93].

What attributes to be identified must be decided according to specific application requirements. When to identify the attributes and how to maintain them is a more difficult decision, which must be based on underlying algorithms, as well as time and space constraints. When comparing the efficiency of two different identification algorithms, we must take into account these specifications of identification.

\section*{3.3 Merging Polygons on a Grid}

\textbf{The Specification}

Dissolving/merging polygons on a grid can be roughly stated as the following requirement. Given a grid and a definition that allows the complete set of polygons to be partitioned into disjoint subsets of \textit{to-be-merged} and \textit{stay} polygons, eliminate the to-be-merged polygons by applying a dissolution/merge operation to each one. This process continues until the result grid contains no to-be-merged polygons. Recall that the dissolution/merge operation is applied to a given polygon, and that the dissolution/merge process is applied to a list of polygons on a grid. However, we still need to specify how the set of to-be-merged polygons is computed and ordered into a sequence to drive the dissolution/merge process, and we need to specify how the dissolution/merge process actually works. Detailed requirements for these aspects of a dissolution/merge process are specified as follows.
We define *aggregating polygons on a grid* (or *aggregation process* in short) as a dissolution/merge process with additional rules that define the following.

1. How is the complete set of polygons in a grid $M$ partitioned into to-be-merged polygons and stay polygons, such that each polygon in $M$ is in exactly one of these partitions?
2. How is the set of to-be-merged polygons ordered into the sequence $<P_1, P_2, ..., P_k>$ which is used to "drive" the dissolution/merge process? The order may be random, from smallest to largest, from largest to smallest, or any order that guarantees that all to-be-merged polygons are eliminated.
3. For the dissolution/merge process itself:
   a. Should a dissolution or merge operation be applied?
   b. How should the target neighbor polygon for each dissolution/merge step be chosen?
      Some possible rules are: choose the neighbor that has the longest common border with the to-be-merged polygon, or choose the neighbor that is most "similar" to the to-be-merged one, where similarity between classes is defined by application requirements.
   c. When the target of a dissolution/merge operation is a to-be-merged polygon, what effects does the dissolution/merge operation have on subsequent process ordering?

**Optimal dissolution/merge process and comparison of algorithms**

If the rule set of an aggregation process is ambiguously or incompletely defined, i.e., the rules in the rule set can be interpreted in ways that result in different to-be-merged set membership, to-be-merged polygon orders, or dissolution/merge operations, we say that some degree of *freedom* exists in the aggregation process. If there is no degree of freedom, the new grid resulting from any implementation of the aggregation process should be always the same. When freedom in the aggregation rules does exist, different *rule implementations* can result in different result grids for the same aggregation process on the same input grid. Many applications do permit freedom in their rule set, so we need criteria to judge between different possible implementations. Comparing different algorithms of different rule implementations is not straightforward. We could use performance characteristics (e.g., time and space costs) or application characteristics. However, different application requirements may need quite different criteria. Without well defined criteria, it is improper to compare different result grids, different rule implementations and different algorithms for the same aggregation process. If we try to find some optimal implementation, optimal criteria need to be well defined.
Any of the GT-Ratios defined in Section 3.1 can be used to compare the grids resulting from two or more different rule implementations for an aggregation process with some degree of freedom. It is possible to define optimality, along the following lines. Assume an aggregation process $A$ based on a merge process $G(M, <P_1, P_2, ..., P_k>) = M'$ and rule set $R$ that permits a set of possible implementations RuleSet-1, RuleSet-2, ..., and RuleSet-m. A result grid $M'$ for aggregation process $A$ is optimal if $M'$ is produced by one of the rule implementations, and for a given criterion, $M'$ is "better" than any other result grid produced by any other rule implementation for $A$. A rule implementation RuleSet-i is optimal if for a given criterion and for all possible input grids, RuleSet-i always produces a "better" result grid than any other rule implementation. Unfortunately it is difficult, or perhaps impossible, to prove related optimality in the general case.

One reason for the difficulty of finding optimal implementations deserves special mention. Assume $M^0$ is the original grid, $M^{i+1}$ is the result grid after $i$th dissolution operation $g(M^i, P) = M^{i+1}$ is applied to $M^i$. Let RuleSet2 be defined such that for each merge step we can produce an optimal $M^i$. We still cannot guarantee that the final result grid produced by RuleSet2 is optimal. The difficulty comes from cascaded effects that occur randomly on a grid, and tend to affect optimality in unpredictable fashion. That is, even if our rule set selects the merge target at each step to produce the "best" result, subsequent cascaded effects can change the result and produce a non-optimal image.

Optimality criteria are still important in practice for the following reasons. When we compare two algorithms, we need to first ask if the two algorithms implement the same rule implementation. Or, when we define an application we need to recognize that a rule implementation may theoretically need underlying algorithms with higher complexity. Thus, when comparing two algorithms for the same aggregation process, both the involved rule implementation and the algorithm specification itself need to be considered. Different applications may choose quite different optimal criteria. In many cases, it may be proper to use "relatively optimal criteria". In practice, a trade-off is usually made between the satisfactory and the optimal. We usually find a satisfactory rule implementation instead of an optimal one. For one input grid, we tend to find a satisfactory new grid without worrying if it is an optimal one. Some kinds of freedom may result in minor differences in the result grids, while others may result in significant differences. When significant differences do occur between rule implementations, the aggregation process may need to be modified to improve the merge result or to avoid some types of dissolution behavior.
3.4 An Example of Rule implementation

In the MMU project described in Section 2.2, the aggregation process is specified as follows. The rule set given below has freedom only in subtle details of handling cascaded merge effects.

(1) All small polygons whose size under a certain threshold are to-be-merged, except for small polygons with "exclusive" class values, which are not to-be-merged.

(2) The to-be-merged polygons are merged in an order from smallest to largest. If two polygons have the same size, the one whose right-most column is smaller is ordered first. If their right-most columns are the same, the one whose top cell on the right-most column has a smaller row number is ordered first.

(3a) Use merger for each polygon.

(3b) The target of each merge operation is determined by selecting the "most similar" neighbor of the source polygon. Similarity is determined by a similarity function: \( \text{Sim}(\text{class}_1, \text{class}_2) \).

(3c) If a polygon resulting from a merge operation is still smaller than the threshold, it remains in its original order on the list of the to-be-merged polygons; if it is now larger than the threshold, it becomes a stay polygon and is logically removed from the to-be-merged list (formally, the merge process for this polygon is "null").

MERGE11 is a program that implements this rule set. MergeRP1 is a program that implements a rule set with a minor modification of rule (2): "if two polygons have the same size, the one whose top row is smaller is ordered first; if their top rows are the same, the one whose right-most cell on the top row has a smaller column number is ordered first". The grids resulting from the two rule sets are generally quite similar. MergeRP3 is program that implements rule set with rules (3b) and (3c) modified. Rule (3b) is modified to: "choose the smallest neighbor as the merge target; if there are more than one neighbor with same minimal size, choose the one with maximum similarity". Rule (3c) is modified to: "if a result polygon is still smaller than the threshold, move it to a point in the list of to-be-merged polygons; appropriate to by its new size". MergeRP2 is a program that implements a rule set with rule (2) of MergeRP1 and with rule (3c) of MergeRP3.
Figure 3.4 Different rule implementations produce new grids with different numbers of remaining polygons.

Figure 3.4 shows how we can use GGT-Ratio as a measurement criterion to compare these rule implementations. orgnl_polys represents the number of polygons in the original grid, stay_polys represents the number of polygons that will not be merged, to-be-merged represents the number of polygons that will be merged, final_polys represents the number of polygons in the final result grid and GGT-R represents GGT-Ratio. Note that "orgnl_polys = to-be-merged + stay_polys". For M1 (100×100) as input grid and a threshold of 10, all the four programs identified the same number of polygons, 6071, partitioned the polygons into the same set of to-be-merged and stay polygons, and merged the 5952 to-be-merged polygons to form a result grid.

Both the number of final polygons and the GGT-Ratio values of MERGE11, MergeRP1 and MergeRP2 are very close. MergeRP3 has both a number of polygons and GGT-Ratio value are significantly greater than those for the other three programs. For M2 (200×200) and threshold 22, the situation is very similar. An intuitive explanation can be given as follows. If we always merge the smallest polygon to its smallest neighbor, the chance of cascaded effect is greatly decreased. Thus, if a polygon is very close to the threshold and it has a small neighbor, then when the small neighbor is merged into it, the target polygon may have its increased size larger than the threshold and thus become a stay polygon.

It is relatively easy to demonstrate that none of the programs is an absolute optimal rule implementation. For example, consider MergeRP3. An counter example is given in Figure 3.5. By the rule implementation of MergeRP3, polygon Q should be merged to P. Because of cascaded effect, P, Q and R form one polygon PQR. size[PQR] = 50 and class_value[PQR] = 4. This is the merge operation from (a) to (b) in Figure 3.4. Another merge operation is from (a) to (c) in the figure. Let G1 be merge process MergeRP3 and G2 is the other merge process.
\( GT_y(G_{1_{M-M_1}}) = 2/4 \) and \( GT_y(G_{2_{M-M_2}}) = 3/4 \). MergeRP3 seems to be a "relatively optimal" rule implementation, but a strict proof of that is difficult to construct.

threshold=22
value means class_value

class_value[Q]=3
class_value[P]=4

(a)

(b)

(c)

Figure 3.5 An example showing that MergeRP3 is not absolutely optimal
4 The Object-Oriented Design Model

This chapter presents an object-oriented design (OOD) model called The Model for Identifying and Merging Raster Polygons (MIMRP), which is developed for the merge problem specified in Chapter 3. The word class used in this chapter represents a concept in OOD philosophy, which has a close relationship to the word object used in Chapter 3, but has nothing to do with the class concept used in Chapters 1 and 2. Both concepts for the word class are used in following chapters and which concept is implied should be clear from the context.

4.1 The MIMRP Model and Its Main Classes

Object-Oriented Design as a new and powerful approach to system design offers hope in revolutionizing the processes of both design and programming. Compared with traditional design approaches, the OOD approach focuses on developing a general description of key data types as objects prior to detailed designing and coding of algorithms. This approach may increase the initial design cost, but it pays dividends in several ways. First and foremost, such extra effort imposes an additional level of discipline on the programming process. Increasing the programmer's discipline almost always pays dividends. Second, the solution is more encapsulated, which makes it more robust, easier to maintain and change, and more reusable [Ira-91]. These advantages of the OOD approach have been confirmed with the MIMRP model, which was applied to the MMU project (see Section 2.2) using C++ as the implementation programming language. It should be noted that although C++, as one of the Objected-Oriented Programming (OOP) languages, is powerful and popular, the OOD approach is independent of any particular programming language. This MIMRP model can be extended, modified and applied to other similar application projects.

The term class is used to represent a group of objects with similar properties (attributes), common behaviors (operations) and common relationships to other objects. An object belonging to
a class is an instance of that class or an class object. The class concept also provides encapsulation, which means all pertinent aspects of the class definition, including its attributes and operations, are packaged together, and the external aspects (accessible to other objects) are separated from the internal implementation details (hidden from other objects). An attribute of a class corresponds to data that an instance of that class will have. An operation is either a class operation or a global operation. An operation of a class is called a class operation. For example, every class must have an instantiation operation, by which class instances are created. A global operation is based on some class operations but does not belong to any class. If X is a class or an object, then an attribute f of X is represented as f[X], and an operation g of X is represented as X → g. An object attribute may be by itself another object which in turn has its own attributes and operations. This is represented as f[f[X]] and f[X] → g2.

The MIMRP model can be described in terms of some specific relationships between a set of classes and global operations. The relationship structure will be described in next section. This section presents the main classes and global operations. It is assumed that the attributes and operations for each class presented here are its external aspects. Its internal attributes and operation implementation details are described in next chapter. Some of the attributes are introduced in Chapter 3, such as class_value[P] and size[P] for polygon P. Names of classes, attributes and operations given here will be frequently referred to. For convenient reference, some names have a short form which is given in parentheses following the formal name. The instantiation operation which every class must have is omitted. The main classes of the MIMRP model are: ImageManager, Image, PolygonList, RuleSet, Polygon, NeighborList and CellSet.

Class Polygon. Figure 4.1 shows attributes and operations of class Polygon, which is one of the most important classes in the MIMRP model. Assume P is a Polygon object. class_value[P] is the class number which is associated to all the cells of P. start_cell[P] can be any cell of P, which is by itself an object with only two attributes i[ start_cell[P] ] and j[ start_cell[P] ], representing the cell’s position <i, j> in grid. In the MIMRP model, start_cell[P] is defined as the left-most cell of the top row in P. start_cell[P] can be used to help decide the order of two polygons with the same size or it can be used as a start point to trace all the other cells in P. original_size[P] is the original size of P before a merge process is applied. During the merge process, following each dissolution operation the target polygon has its size increased by absorbing the source polygon. final_size[P] is used to dynamically maintain the size of P during the merge process, so that when the process is completed, it is the final size of the polygon. After P is dissolved, it logically ceases to exist, and thus becomes a dead polygon. This is indicated by assigning a negative number to its final_size.
neighbor_list[P] is used to maintain the neighbor information for P, in the form of an object of class NeighborList. Similarly, cell_set[P] is an object of class CellSet that maintains locational information of every cell of P. Operation P→Add-Cells is used to add one or more cells to P. P→Add-Neighbor adds a neighbor to neighbor_list[P]. P→Merge-In is used to absorb a neighbor polygon, i.e., to implement a merge operation. P→Output writes all the cells of P to a result grid. P→Report reports the status of P, such as class_value[P], original_size[P] and final_size[P].

<table>
<thead>
<tr>
<th>Attributes:</th>
<th>Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_value; start_cell (str_cell); original_size (orgnl_size); final_size (fnl_size); neighbor_list (nlist); cell_set;</td>
<td>Add-Cells; Add-Neighbor (Add-Neibr); Merge-In; Output; Report;</td>
</tr>
</tbody>
</table>

Figure 4.1 Class Polygon

**Class PolygonList.** Figure 4.2 shows attributes and operations of class PolygonList, which represents a list of Polygon objects. If L is a PolygonList, head[L] and tail[L] are the first polygon and last polygon on the list, respectively. total_count[L] is the total number of polygons on the list and dead_count[L] is the number of dead polygons on the list. L→Head-Get returns the first polygon on the list. L→Tail-Add is used to add a new element to the polygon list at the end. L→Merge-In merges another PolygonList object with L. L→Release is used to delete all the polygons on the list and free the memory associated with those polygons. L→Report reports the status of L.

In the MIMRP model, lists of to-be-merged polygons and stay polygons are objects of class PolygonList.

<table>
<thead>
<tr>
<th>Attributes:</th>
<th>Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>head; tail; total_count; dead_count;</td>
<td>Tail-Add; Head-Get; Merge-In; Release; Report;</td>
</tr>
</tbody>
</table>

Figure 4.2 Class PolygonList
**Class CellSet.** Figure 4.3 shows attributes and operations of class CellSet, which is designed to maintain locational information for all the cells of a Polygon object. Each cell has a row number and a column number to indicate its position in grid. Assume C is a CellSet object and P is a Polygon object. start_row[C] is the top row of C, i.e., the smallest row number of all cells in C, and end_row[C] is the bottom row of C, i.e., the largest row number of cells in C. size[C] is the number of cells contained in C. Operation CellSet→Add-Cells adds some cells to a CellSet object. It is mainly called by operation Polygon→Add-cells. CellSet→Release is used to delete all the cells and release the memory. This operation is used when a CellSet object is merged into another one. C→Merge-In adds all the cells in another CellSet object into C. C→Report reports the status of C, such as start_row[C] and size[C].

<table>
<thead>
<tr>
<th>Class CellSet</th>
<th>Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes:</td>
<td></td>
</tr>
<tr>
<td>start_row (str_row);</td>
<td>Add-Cells;</td>
</tr>
<tr>
<td>end_row;</td>
<td>Merge-In;</td>
</tr>
<tr>
<td>size;</td>
<td>Release;</td>
</tr>
<tr>
<td></td>
<td>Report;</td>
</tr>
</tbody>
</table>

Figure 4.3 Class CellSet

**Class Image.** Figure 4.4 shows attributes and operations of class Image. An Image object contains a grid plus additional attributes. Assume M is an Image object. header[M] contains information such as the data type for class numbers, the count of class numbers and some other features of a grid. mode[M] is used to indicate an original image or a new image resulting from a merge process. row_max[M] and col_max[M] represent the number of rows and the number of columns in the grid, respectively. Image→Read-Data reads in grid data from an image file and Image→Write-Data writes out the new result grid to an image file. Image→Report reports status of an Image object.

<table>
<thead>
<tr>
<th>Class Image</th>
<th>Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes:</td>
<td></td>
</tr>
<tr>
<td>header;</td>
<td>Read-Data;</td>
</tr>
<tr>
<td>name;</td>
<td>Write-Data;</td>
</tr>
<tr>
<td>row_max;</td>
<td>Report;</td>
</tr>
<tr>
<td>col_max;</td>
<td></td>
</tr>
<tr>
<td>mode;</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4 Class Image
**Class ImageManager.** This is another important class in the MIMRP model. Figure 4.5 shows its attributes and operations. Assume IM is an ImageManager object. old_image[IM] is the original image and new_image[IM] is the result image when the merge process is completed. rule_set[IM] is an object of class RuleSet, which specifies the merge rules. stay_plist[IM] maintains the stay polygons and tbm_plist[IM] maintains the to-be-merged polygons. These two polygon lists are dynamically changed during the merge process. orgnl_polys[IM], stay_polys[IM] and tbm_polys[IM] provides the number of original polygons, number of stay polygons and number of to-be-merged polygons, respectively, before the merge process. final_polys[IM] gives the number of polygons in the result grid. ImageManager has the three most important operations in the MIMRP model, which will be implemented as algorithms in next chapter. IM \(\rightarrow\) Identify-Polys identifies concerned polygons in the original grid, IM \(\rightarrow\) Merge-Polys is the operation for the merge process and IM \(\rightarrow\) Build-New-Grid writes out the final polygons on the stay_plist to a new image file. IM \(\rightarrow\) Final-Report reports data such as orgnl_polys[IM], stay_polys[IM], tbm_polys[IM], final_polys[IM], and optionally, status of each stay polygon in the result image.

<table>
<thead>
<tr>
<th>Class ImageManager</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attributes:</strong></td>
</tr>
<tr>
<td>old_image;</td>
</tr>
<tr>
<td>new_image;</td>
</tr>
<tr>
<td>rule_set;</td>
</tr>
<tr>
<td>orgnl_polys;</td>
</tr>
<tr>
<td>stay_polys;</td>
</tr>
<tr>
<td>tbm_polys;</td>
</tr>
<tr>
<td>final_polys;</td>
</tr>
<tr>
<td>stay_poly_list (stay_plist);</td>
</tr>
<tr>
<td>tbm_poly_list (tbm_plist);</td>
</tr>
<tr>
<td><strong>Operations:</strong></td>
</tr>
<tr>
<td>Identify-Polys;</td>
</tr>
<tr>
<td>Merge-Polys;</td>
</tr>
<tr>
<td>Build-New-Grid;</td>
</tr>
<tr>
<td>Final-Report;</td>
</tr>
</tbody>
</table>

Figure 4.6 Class ImageManager

**Class NeighborList.** This is a class that is very similar to class PolygonList. In OOD philosophy, a subclass inherits all the attributes and operations from a superclass with optional modification and addition. Thus, in the MIMRP model class NeighborList is treated as subclass of PolygonList, which is the superclass in this inheritance relationship. Special attributes and operations of NeighborList will be introduced in next chapter when necessary.
Class RuleSet. This class is designed to encapsulate a set of aggregation rules, as defined in Section 3.3. Its attributes and operations are dependent on individual applications. The two main operations of RuleSet are RuleSet→Partition-Poly, which decides if a polygon will be to-be-merged or not-to-be-merged, and RuleSet→Find-Target, which decides the target neighbor polygons for each dissolution or merge operation.

Global operations. Add-Neighbors (Add-Neibrs) adds neighbor information for two neighboring polygons, i.e., information on neighbors a and b is added symmetrically to b and a. Polygon-Merge0 (Poly-Merge0) and Polygon-Merge1 (Poly-Merge1) are used to merge two neighboring polygons together to form a single result polygon. These global operations and a few more will be described as algorithms in next chapter.

4.2 Relationship Structure of the MIMRP Model

Figure 4.7 shows the relationships between the classes and global operations described in last section. Most notation and concepts used here for describing OOD models are adopted from [Rumbaugh-91]. The sign ◦ indicates an aggregation relationship between classes, which means a class object as an entire assembly consists of other class objects as its components. Dots and numbers near to it indicate multiplicity, which means how many instances of one class may relate to a single instance of an associated class. The triangle Δ indicates an inheritance relationship, which is between a class and one or more refined versions of it. The class being refined is called the superclass and each refined version is called a subclass. A global operation uses arrow lines to point to the involved classes.

In the diagram, ImageManager has an aggregation relationship with classes Image, PolygonList and RuleSet. The multiplicity shows that an ImageManager object may have one RuleSet object, at least two Image objects (one for the original image and another for the result image for each image process), and at least two Polygon objects (one for the to-be-merged polygons and another for the stay polygons). Class PolygonList has an aggregation relationship with class Polygon and the multiplicity shows that one PolygonList object should have one or more Polygon objects. Class Polygon has an aggregation relationship with classes CellSet and NeighborList. The multiplicity shows that each Polygon object should have exactly one CellSet object and one NeighborList object. There is an inheritance relationship between class Polygon and
class NeighborList. All the three global operations are related to class Polygon, which means these operations will take Polygon objects as their parameters.

Figure 4.6 The relationship structure of the MIMRP model
5 The Algorithms

This chapter describes algorithms designed to implement the key parts of a rule-based aggregation process. The three main algorithms are: one to identify all polygons on a grid, one to implement a rule-based merge process, and one to build the result grid from polygon attributes. Classes and operations of the MIMRP model specified in Chapter 4 are referred to in this chapter. An algorithm can also be considered an implementation of the corresponding operation. The words algorithm and operation are sometimes used interchangeably in this chapter and their meanings should be clear from the context. Some new classes and operations are also specified in detail here. Each algorithm is described in "pseudocode" in a figure. Simple operations are only explained in English without detailed algorithm specification.

5.1 The Pseudocode, Classes and Operations

The Pseudocode and conventions

An algorithm can be specified as a program in a specific language. For example, the algorithms given in this chapter have been implemented in C++. In fact, a programming language reflects, to a large extent, the language developer's understanding of some algorithm design principles. Thus an algorithm is often bound up with language-specific details. However, people of different programming language backgrounds need to share understanding of algorithms, and for a complicated algorithms, the essence of the algorithm needs to be extracted from program details. Thus, pseudocode is introduced. Different textbooks introduce different pseudocodes [Cormen-90, Bach-86]. Although we may express an algorithm in any well-defined pseudocode, some pseudocodes may be more convenient for some problems. This is like translating programs between different languages. Usually, a pseudocode is based on a particular language, with modifications
made to reduce the language dependent detail and to improve readability. What separates
pseudocode from "real" code is that, in pseudocode, a more clear and concise expressive method is
employed to specify a given algorithm. Thus sometimes an English phrase, sentence or paragraph
is embedded in a section of the pseudocode description in place of programming detail. Issues such
as data type definition and error handling can be ignored in order to convey the essence of the
algorithms more concisely.

The pseudocode used in this chapter is based on the syntax of C++. The class construct of C++
is in fact an extended implementation tool for abstract data types (ADTs) and can be directly used
to implement concepts class and object introduced in Chapter 4. Since C++ is not as readable as
Ada and Modula-2, some modifications are made and some notation is borrowed from these
languages. Thus, and is used instead of &&, or is used instead of ||, parameter passing by value or
by reference is explicitly indicated by in and in_out, like what is used in Ada. If X is a class or an
instance of a class, f[X] is used to represent an attribute (or a field, in programming language
terms) of X. An operation of X is represented as X->f. Different classes can have operations with
the same name. The operations are distinguished by the prefixed class name. A class operation may
be used without the prefix when no confusion will occur in the local context. Global operations are
used without a class name prefix. Within specification of an algorithm for an operation of class X,
attributes of X can be directly referred to without having to specify the suffix [X]. Names of
algorithms are printed in small capitals like POLY-SORT. Names of simple operations are printed
regularly, like ImageManager->Partition-Poly. In a piece of pseudocode, when English sentences
are used to describe a function, the sign > is prefixed. Any one who is familiar with C/C++, Ada,
Pascal or Modula-2 should have little trouble reading the algorithm descriptions. In the algorithm
analysis, O( ) means time complexity when complexity analysis is being discussed. For simplicity,
we assume a "n x n" grid.

Classes and operations

The three main algorithms are specified for the three operations of the MIMRP model:
ImageManager->Identify-Poly, ImageManager->Merge-Polys and ImageManager->Build-New-
Grid. A complicated algorithm is usually broken up into small ones. Classes and operations are
introduced here to help describe internal implementation details of the MIMRP model. Figure 5.1
shows a polygon P and its representation in the MIMRP model. Each row of cells in P is
represented as a list of integer pairs. Such an integer pair is called a column run (or just run).
Comparing this type of representation with what is described in the MMU project (see Figure 2.2,
Section 2.2), the difference is that here each integer pair is a column run instead of a "row run". By the raster polygon definition, a cell in a grid can be in only one polygon. Thus, the runs describing two different polygons cannot have an intersection.

![Figure 5.1 A polygon and its representation in column runs](image)

Class ColRunList, as shown in Figure 5.2-a, is introduced to describe the column run representation of polygon. The attribute col_run_vect[ColRunList] maintains a list of column runs. ColRunList→Add-Cells adds a column run in the list. ColRunList→Output writes a class value to a vector of class number data type for those cells represented by col_run_vect[ColRunList]. Class RowList, as shown in Figure 5.2-b, describes all the runs for a polygon, in terms of a list of ColRunList instances. It is in fact an implementation of class CellSet specified in Chapter 4. From now on, attribute cell_set[Polygon] is considered to be a reference to an instance of RowList. Attribute clist_vect[RowList] maintains a list of ColRunList instances for a polygon. str_row[RowList] corresponds to str_row[CellSet], and end_row[RowList] corresponds to end_row[CellSet]. RowList→Add-Cells adds a column run at a specific row by calling ColRunList→Add-Cells.

![Figure 5.2 Classes ColRunList and RowList](image)
As illustrated in Figure 5.3, RowList→OUTPUT is used to write out a specified range of rows to an array using cls_value, which is the class value of the polygon. n1 and n2 specify a range in grid. out_array is an array with size of col_max[Image] columns and \((n2 - n1 + 1)\) rows, which is used to output cells in clist_vect[RowList] to the result grid. done is used to indicate if the polygon has all its cells written out. This operation will be used in the algorithm for building the new result grid.

```plaintext
Algorithm RowList→OUTPUT(in_out out_array, done, in cls_value, n1, n2)
// n1-n2 is a specified range of rows in grid
// precondition: str_row >= n1;

if (str_row < n1 or str_row > end_row)
  ▷ error: this should not happen.

if (str_row > n2) { // no cells in the specified rows
done = false;
return;
}

if (end_row <= n2) {
  ▷ output all the rows from str_row to end_row by
  using ColRunList→Output;
  str_row = str_row + n2 - n1 + 1;
  if (str_row > end_row) {
    done = true;
    return;
  }
}
else {
  ▷ output rows from str_row to n2, using ColRunList→Output;
  str_row = n2 + 1;
  done = false;
  return;
}
```

Figure 5.3 Algorithm RowList→OUTPUT

The global operation CList-QMerge, as shown in Figure 5.4, does a quick merge of two ColRunList instances by simply adding clist2 to the end of clist1 and returning the merged result new_clist. CList-GMerge, also shown in Figure 5.4, is a graceful version of CList-QMerge, which re-organizes the result new_clist such that the column runs are ordered and "neighboring" runs are combined. Obviously, the time complexity of CList-QMerge is \(O(1)\) and CList-GMerge has the time complexity \(O(n)\).
clist1: (1 2) (4 4) (7 9)
clist2: (3 3) (5 6) (12 14)

CList-QMerge(clist1, clist2) returns new_clist as

new_clist: (1 2) (4 4) (7 9) (3 3) (5 6) (12 14)

CList-GMerge(clist1, clist2) returns new_clist as

new_clist: (1 3) (4 9) (12 14)

Figure 5.4 An example for CList-QMerge and CList-QMerge

Algorithm RLIST-MERGE0, illustrated in Figure 5.5, implements a global operation RList-Merge0. This operation merges two RowList instances, rlist1 and rlist2 and returns a RowList instance that is either rlist1 or new_rlist. This is particularly designed for the polygon identification algorithm by taking advantage of the condition that two polygons have the same bottom row when the merger operation occurs, i.e., end_row[rlist1] = end_row[rlist2]. row_i is the currently scanned row when the merger occurs and also the bottom row of both rlist1 and rlist2. The precondition is that the two polygons have been sorted by POLY-SORT (see Figure 5.7) so that top row of rlist1 is not greater than rlist2. Non-intersection rows of rlist1 remain unchanged or are copied directly into new_rlist. If rlist2 is a newly created RowList instance at row_i, then rlist1 and rlist2 only intersect at row_i and are merged with only one operation of CList-Merge/CList-GMerge. The result is put into rlist1 and it is returned while rlist2 is destroyed. For other cases, more than one intersection row exists. RowList(str_i) instantiates a RowList instance with str_i as the start row. new_rlist is used to build result, and both rlist1 and rlist2 are destroyed.

The time complexity of this algorithm depends on whether CList-QMerge or CList-GMerge is used for merging the two ColRunList instances from rlist1 and rlist2. If CList-QMerge is used, the complexity is \(O(n)\); otherwise it is \(O(n^2)\).
Algorithm RLIST-MERGE0\(\text{in } \text{row}_i, \text{in\_out } \text{rlist}_1, \text{rlist}_2\)
\text{return } \text{RowList}\]
// precondition: \text{str\_row}[rlist] <= \text{str\_row}[rlist2]
// \text{row}_i is the currently scanned row

\text{end\_row}_1 = \text{end\_row}[\text{rlist}_1];
\text{end\_row}_2 = \text{end\_row}[\text{rlist}_2];

\text{if} (\text{row}_i \neq \text{end\_row}_1)
  \quad \triangleright \text{ increase } \text{end\_row}[\text{rlist}_1] \text{ by } 1
\text{if} (\text{row}_i \neq \text{end\_row}_2)
  \quad \triangleright \text{ increase } \text{end\_row}[\text{rlist}_2] \text{ by } 1

\text{if} (\text{end\_row}_2 = \text{str\_row}[\text{rlist}_2]) \{ // a special case
  \triangleright \text{ merge two polygons' bottom rows, using CList-QMerge/CList-GMerge}
  \text{return } (\text{rlist}_1); \}

\text{new\_rlist} = \text{new } \text{RowList}(\text{str}_i);
\text{str}_i = \text{str\_row}[\text{rlist}_1];
\text{k} = \text{str\_row}[\text{rlist}_2];
\triangleright \text{ directly copy rows from } \text{str}_i \text{ to } \text{k}-1 \text{ in } \text{rlist}_1 \text{ to } \text{new\_rlist};
  \quad \text{using CList-QMerge/CList-GMerge, merge rows from } \text{k} \text{ to } \text{row}_i \text{ in } \text{rlist}_1
  \text{and } \text{rlist}_2 \text{ and put the result into } \text{new\_rlist}.
\text{return } \text{new\_rlist};

Figure 5.5 Algorithm RLIST-MERGE0

Algorithm RLIST-MERGE1 is shown in Figure 5.6. Two RowList instances \text{cell\_set}_1 \text{ and } \text{cell\_set}_2 \text{ are merged into } \text{new\_cell\_set} \text{ as the returned result. This algorithm, which is a more}
\text{general version of RLIST-MERGE0, is for merging two neighboring polygons during the merge}
\text{process. The } \text{new\_cell\_set} \text{ is divided into three sections: top non-intersection rows, intersection}
\text{rows in the middle and bottom non-intersection rows. The intersection rows of two polygons are}
\text{merged and put into } \text{new\_cell\_set} \text{. Non-intersection rows are copied directly into the } \text{new\_cell\_set} \text{. If there are no intersection rows for}
\text{cell\_set}_1 \text{ and } \text{cell\_set}_2, \text{ end\_row}[\text{cell\_set}_1] \text{ and }
\text{str\_row}[\text{cell\_set}_2] \text{ must have difference of } 1 \text{. This can be used for error checking.}

\text{The time complexity analysis of this algorithm is similar to that of RList-Merge0. If CList-}
\text{Merge0 is used, the time complexity is } O(n); \text{ otherwise it is } O(n^2).
Algorithm RLIST-MERGE1(in_out cell_set1, cell_set2)
return RowList
// precondition: str_row[cell_set1] <= str_row[cell_set2]

str_i = str_row[cell_set1];

// find the intersection
k1 = str_row[cell_set2];
if (end_row[cell_set1] < end_row[cell_set2]) {
k2 = end_row[cell_set1];
end_i = end_row[cell_set2];
}
else {
k2 = end_row[cell_set2];
end_i = end_row[cell_set1];
}

new_cell_set = new RowList(str_i);
> directly copy rows from str_i to k1-1 in cell_set1 into new_cell_set;
> merge each row from k1 to k2 in cell_set1 and cell_set2 and copy
the result into new_cell_set;
> copy rows from k2+1 to end_i in cell_set1 or cell_set2, depending on
which has the lower end_row[RowList], into new_cell_set;
return new_cell_set;

Figure 5.6 Algorithm RLIST-MERGE1

Algorithm POLY-SORT is shown in Figure 5.7, which implements a simple global operation. This algorithm sorts two polygon instances poly1 and poly2 by using their str_cells. When the
algorithm is completed, poly1 is the "smaller" one, which is determined as follows: choose the one
whose top row is smaller; if their top rows are the same, choose the one whose left-most cell on the
top row has a smaller column number.

Algorithm POLY-SORT(in_out poly1, poly2)
if (i(str_cell[poly2]) < i(str_cell[poly1]) or
    (i(str_cell[poly2]) == i(str_cell[poly1]) and
     j(str_cell[poly2]) < j(str_cell[poly1])) {  
> switch poly1 and poly2
    }

Figure 5.7 Algorithm POLY-SORT
5.2 The Identification Algorithm

The algorithms which are directly called by the algorithm ImageManager→IDENTIFY-POLYS are introduced first.

Algorithm Manager→LOOK-UP is shown in Figure 5.8. For the currently scanned cell at <i, j> in a grid, this algorithm looks up its left cell (l_dat), upper cell (u_dat), upper-left cell (lu_dat), right cell (r_dat), the left polygon (l_id) that contains the left cell, and the upper polygon (u_id) that contains the upper cell. When upper and left polygons are actually the same polygon, they are merged and one of them becomes a dead polygon. In ImageManager→LOOK-UP, if P is a dead polygon, the attribute to-other[P] refers to the target polygon that actually absorbed it. Because of the way the links are maintained, the maximum number of links from a dead polygon to an active polygon is 2. The while loop is added to guarantee that each time the number of iterations for finding the eventual active polygon is bounded by 2. That is, if to-other[P] is a dead polygon then to-other[to-other[P]] must not be a dead polygon because each time to-other[P] is updated in the while loop.

Algorithm ImageManager→LOOK-UP

\[
\begin{align*}
\text{r_dat} &= \text{dat_cache}[i \% 3][j + 1]; \\
\text{l_dat} &= \text{dat_cache}[i \% 3][j - 1]; \\
\text{u_dat} &= \text{dat_cache}[(i - 1) \% 3][j]; \\
\text{lu_dat} &= \text{dat_cache}[(i - 7) \% 3][j - 7]; \\
\text{l_id} &= \text{id_cache}[i \% 3][j - 7]; \\
\text{u_id} &= \text{id_cache}[(i - 1) \% 3][j]; \\
\end{align*}
\]

if \((\text{l_id} \neq \text{null} \text{ and to-other}[\text{l_id}] \neq \text{null})\) { 
  while \((\text{to-other}[\text{to-other}[\text{l_id}]] \neq \text{null})\) 
    \(\text{to-other}[\text{l_id}] = \text{to-other}[\text{to-other}[\text{l_id}]];\) 
    \(\text{l_id} = \text{to-other}[\text{l_id}];\) 
  } 

if \((\text{u_id} \neq \text{null} \text{ and to-other}[\text{u_id}] \neq \text{null})\) { 
  while \((\text{to-other}[\text{to-other}[\text{u_id}]] \neq \text{null})\) 
    \(\text{to-other}[\text{u_id}] = \text{to-other}[\text{to-other}[\text{u_id}]];\) 
    \(\text{u_id} = \text{to-other}[\text{u_id}];\) 
  }

Figure 5.8 Algorithm ImageManager→LOOK-UP
Algorithm **ADD-NEIBRS** is shown in Figure 5.9, in which *threshold* is the threshold size used to determine which polygons are to-be-merged, and *poly1* and *poly2* are two neighboring polygons. For *poly1*, if *size[*poly1]* is less than the *threshold*, *poly2* will be added to *poly1*'s the neighbor list *nlist[*poly1]* by calling *poly1→Add-Neibr(poly2)*, which simply adds *poly2* to the end of *nlist[*poly1]* by calling *NeighborList→Tail-Add*. For *poly2*, the process is similar. We could implement *Polygon→Add-Neibr* to check and eliminate duplicate neighbors, but the cost of checking is too high. This issue will be addressed later when specifying algorithm *ImageManager→IDENTIFY-POLYS*.

```
Algorithm ADD-NEIBRS(in threshold, in_out poly1, poly2)
// if size of a polygon is under the threshold, its neighbors are recorded;
// otherwise not. Polygon→Add-Neibr adds a neighbor to a polygon's
// neighbor list.
if (*size[*poly1]* <= threshold)
  poly1→Add-Neibr(poly2);
if (*size[*poly2]* <= threshold)
  poly2→Add-Neibr(poly1);
```

Figure 5.9 Algorithm **ADD-NEIBRS**

Algorithm **POLY-MERGE0**, as illustrated in Figure 5.10, implements the global operation Poly-Merge0 in the MIMRP model. This is the most important global operation for the polygon identification algorithm. *Poly1* and *Poly2* are two neighboring polygons to be merged at row *row_i*. *inter_plist* is a *PolygonList* instance for maintaining dead polygons for the processes of both identification and merge. First *poly1* and *poly2* are sorted by using algorithm **POLY-SORT** (see Figure 5.7) so that *poly1* is the "smaller" one, i.e., *str_row[poly1]* ≤ *str_row[poly2]*. After **POLY-SORT** is applied, *poly1* is used as the target polygon for the merge operation. *size[Polygon]* is an attribute introduced to dynamically record the size of a polygon, i.e., in **POLY-MERGE0**, *size[*poly1]* is increased by adding *size[*poly2]*. Also, *poly2*'s neighbor list is added to end of *poly1*'s neighbor list by calling *NeighborList→Merge-In*, and *cell_set[*poly2]* is moved into *cell_set[*poly2]* by calling *RLIST-MERGE0*. *P2* is marked as a dead polygon by assigning −1 to *size[*P2]*, and it is put into the *inter_plist*. *poly1* is returned as the result polygon from the merge operation.
Algorithm POLY-MERGE0( in row_i, in_out polyl, poly2, inter_plist)

return Polygon

// inter_plist[ImageManager] is a polygon list for those "dead" polygons

sl = size[poly1];
s2 = size[poly2];
POLY-SORT(poly1, poly2);  // now poly1 is the target polygon
size[poly1] = sl + s2;
nlist[poly1]→Merge-In(nlist[poly2]);  // add neighbors of poly2 to poly1's neighbor list
cell_set[poly1] = RLIST-MERGE0(row_i, cell_set[poly1], cell_set[poly2]);
size[poly2] = -1;  // indicating a dead polygon
to-other[poly2] = poly1;
inter_plist→Tail-Add(poly2);

return (poly1);

Figure 5.10 Algorithm POLY-MERGE0
Algorithm \textit{ImageManager}→\textsc{Identify-Polys()}

\begin{verbatim}
for (i = 1; i <= row_max; i++) {
  ii = i % 3; // ii is used to index dat_cache[]
  old_image→Read-Data(dat_cache[?], col_max); // col_max = col_max[old_image]

  // read one row of old grid from the image file into dat_cache[?]
  c1 = 1;

  for (j = 1; j <= col_max; j++) {
    pix = dat_cache[ii][j]; // pix is the currently processed cell
    ImageManager→Look-Up(i, j, l_dat, l_id, u_dat, u_id, lu_dat, r_dat);
      // l_dat: left cell, l_id: left polygon, u_dat: upper cell, u_id: upper polygon,
      // lu_dat: upper-left cell, r_dat: right cell.
    if (pix == l_dat and pix == u_dat) { // case 1, see Figure 5.12
      ... (code)
    } else if (pix != l_dat and pix != u_dat) {
      ... (code)
    } else if (pix == l_dat) { // case 2, see Figure 5.13
      ... (code)
    } else { // case 3, see Figure 5.14
      ... (code)
    }
    if (r_dat != pix) {
      ... (code)
    }
    if (i > 2)
      rule_set→Partition-Polys(id_cache[(i-2) % 3][j]);
        // put the polygon into tbm_plists or stay_plist
  }
}
\end{verbatim}

Figure 5.11 Algorithm \textit{ImageManager}→\textsc{Identify-Polys}
Algorithm ImageManager→IDENTIFY-POLYS, as illustrated in Figure 5.11, implements the class operation ImageManager→Identify-Polys in the MIMRP model (refer to Section 4.1 for the description of class ImageManager). The grid in old_image is scanned one row at a time from top to bottom using old_image→Read-Data. For each row, cells are then processed one by one from left to right. pix is used for the currently processed cell. For each pix, its left cell (l_dat), upper cell (u_dat), upper-left cell (lu_dat), right cell (r_dat), the left polygon (l_id) that contains the left cell, and the upper polygon (u_id) that contains the upper cell are each identified by calling ImageManager→LOOK-Up. There are four cases that determine to which polygon (l_id or u_id) the cell pix should belong and how to add neighbor information for l_id, u_id and current polygon id. In Figures 5.12 to 5.15, the shaded cell is the current one and the wild card character * in a cell represents any class value.

Case 1, as illustrated in Figure 5.12-a, occurs when upper cell, left cell and current cell are the same. This means that upper polygon and left polygon are actually the same polygon (u_id = l_id) and current cell belongs to this polygon. If l_id and u_id are not the same, as shown in Figure 5.12-c, then one of them is merged into the other to form one polygon, using POLY-MERGE0. If l_id and u_id are the same, as shown in Figure 5.12-b, no such merge operation occurs and the left polygon l_id is assigned to id.

Case 2, as illustrated in Figure 5.13-a and Figure 5.13-b, occurs when both left cell and upper cell are not the same as current cell. This means a new polygon should be created for current cell and neighbor information should be properly added to id, l_id, and u_id. If l_id equals u_id, as shown in Figure 5.13-c, the two neighbors are parts of the same polygon, so only l_id and id need to add the neighbor information to each other. Otherwise as shown in Figure 5.13-d, neighbor information needs to be added between l_id and id, and between u_id and id. The global operation ADD-NEIBRS adds the neighbor information for two neighboring polygons.
Case 3, as shown in Figure 5.14-a, occurs when left cell is the same as the current one. Thus \( l_id \) is assigned to current polygon \( id \). If both left cell and upper cell are not equal to upper-left cell, as shown in Figure 5.14-b, neighbors are added to each other for \( u_id \) and \( id \). In Figure 5.14-c and Figure 5.14-d, the ADD-NEIBR operation is not needed because the neighbor relationship was added when \( u\_dat \) and \( l\_dat \) were being processed.

Case 4, as illustrated in Figure 5.15, occurs when upper and current cells are the same. This case is symmetric to Case 3.

When the status of the current cell has been decided and neighbor relationships have been properly added for the three polygons \( id \), \( l_id \) and \( u_id \), the algorithm checks to see if a column entry has been found for the current polygon \( id \), by comparing current cell and right cell. If they are not same, a column run \(<c1, c2>\) is found and it is added to current polygon \( id \). Following the processing of all cells on row \( k \), \( k \geq 2 \), we can classify all polygons containing elements on row \( k - 1 \), as either bounded or unbounded. A polygon with an entry on row \( k - 1 \) is bounded (i.e., its complete size and extent are known) if it contains no entries on row \( k \); a polygon with an entry on
row $k - 1$ is unbounded if it also has an entry on row $k$ (i.e., its size and extent can still grow). Thus, at this point, for all bounded polygons we can classify them into the list of stay polygons $stay\_plist$, or the list of to-be-merged polygons $tbm\_plists$ that is a vector of $PolygonList$ instances. This operation of checking whether a polygon is "bounded" and classifying it accordingly is done by $rule\_set\rightarrow Partition\_Polys$. For each bounded polygon $P$, if $P$ belongs to the to-be-merged list and $size[P]$ is $i$, then $rule\_set\rightarrow Partition\_Polys$ adds $P$ to the end of $tbm\_plists[i]$. Because of the order in which the polygons are identified, when $IDENTIFY\_POLYS$ is completed, all the to-be-merged polygons are already sorted. And if to-be-merged polygons $P_1$ and $P_2$ have the same size, they are automatically ordered in the order specified in algorithm $POLY\_SORT$. At the end of input image scanning, polygons for the last two rows need to be checked and classified by $Partition\_Polys$. To avoid classifying a polygon more than once, a tag for each polygon is used.

**Complexity analysis.** Let $IMI$ be the short form of $ImageManager\rightarrow IDENTIFY$. Since each cell must be processed once, the best case is $\Omega(IMI) = \Omega(n^2)$, where the $\Omega(f(n))$ is another notation of complexity to measure the lower bound of an algorithm. Figure 5.16 shows the calling relationship between $ImageManager\rightarrow IDENTIFY\_POLYS$ and other operations which are called. If all the called operations within $IDENTIFY\_POLYS$ have time complexity $O(1)$, $O(IMI)$ is $O(n^2)$. The operation $POLY\_MERGE0$ is the only one whose time complexity is not $O(1)$. If time complexity of $NeighborList\rightarrow Merge\_In$ and $RL\_LIST\_MERGE0$ within $POLY\_MERGE0$ are both $O(1)$, $POLY\_MERGE0$ is $O(1)$. We assume that $NeighborList\rightarrow Merge\_In$ simply adds another $NeighborList$ instance at the end, thus it has time complexity $O(1)$. The time complexity of $RL\_LIST\_MERGE0$ is $O(n)$ if $CL\_LIST\_Q\_Merge$ is used, or $O(n^2)$ if $CL\_LIST\_G\_Merge$ is used. Thus $O(IMI)$ is $O(n^2)$ if $CL\_LIST\_Q\_Merge$ is used within the algorithm. Note that $O(n^3)$ is the worst case. By adding special processing for some special cases, the time complexity is very close to $O(n^2)$ in practice. The space complexity of $IDENTIFY\_POLYS$ is $O(n^2)$.

Although the chance of adding duplicate neighbors has been greatly decreased by distinguishing the four cases in $ImageManager\rightarrow IDENTIFY\_POLYS$, a neighbor may still be added more than once. Having duplicate neighbors increases the time and space complexity of algorithm $ImageManager \rightarrow MERGE\_POLYS$. We could have implemented $Polygon\rightarrow Add\_Neibr$ and $NeighborList\rightarrow Merge\_In$ gracefully such that duplicate neighbors are not added, but the cost would be too high because each time a neighbor is added the whole neighbor list needs to be searched. This greatly increases the time complexity of $IMI$. Our judgment is that the improvement resulting from eliminating all duplicate neighbors is not justified by the increased cost in $Polygon\rightarrow Add\_Neibr$ and $NeighborList \rightarrow Merge\_In$. 
Figure 5.16 The calling relationship between `ImageManager.Identify-Polys` and other operations
5.3 The Algorithm for the Merge Process

Algorithm ImageManager→MERGE-POLYS, as shown in Figure 5.17, implements the operation ImageManager→Merge-Polys in the MIMIRP Model. After ImageManager→IDENTIFY-POLYS is completed, all polygons are identified and put on stay_plist and tbm_plists, each tbm_plists[i] stores the to-be-merged polygons with size equal to i. Recall that the identification algorithm places all polygons in each tbm_plists[i] in the correct order, so that no sorting is necessary at this step. To-be-merged polygons are processed one by one from the smallest to the largest, according to their order on tbm_plists. Each to-be-merged polygon is first checked to see if its size has increased to a value greater than the threshold, i.e., because it has been a target neighbor for one or more merge operations. If so, the polygon is moved to the stay polygon list stay_plist. If the polygon's size is smaller than the threshold, the target neighbor target_neibr is found according to the given merge rule. Because of cascaded effects, we also need to check for neighbors of target_neibr whose class value is equal to class_value[target_neibr]. Such neighbors are stored in casc_neibrs, which is a PolygonList object. Then target_neibr, poly and polygons in casc_neibrs are all merged together using operation Poly-Merge1. Poly-Merge1 is almost the same as POLY-MERGE0, except that within Poly-Merge1, RLList-MERGE1 is used instead of RLList-MERGE0. When all mergers are done, one of the polygon becomes the result polygon result_poly. If it is a stay polygon, it remains in the stay polygon list. If it was originally on the list of to-be-merged polygons and has a new size less than the threshold, it stays where it was; if its size is greater than than the threshold, it is moved from tbm_plists to stay_plist. Note we do not reposition the result_poly on the to-be-merged list by its new size, although this is an option which could be implemented.

Complexity analysis. tbm-polys[ImageManager] is the number of to-be-merged polygons. Let $N_m$ be the number of merge operations which occur during the whole merge process, including those because of cascaded effects, and $U$ be the average size of to-be-merged polygons. The upper bound of $N_m$ is $O(N_m) = K \times (N_n / 2)$, where $K$ is the number of to-be-merged polygons and $N_n$ is the maximum number of neighbors for any to-be-merged polygon. Since the cascaded effects occur randomly in a grid, $N_m$ cannot be known beforehand. In general, the greater tbm-polys is, the more chances for cascaded effects, and thus the greater $N_m$ is. However, there are some cases where a smaller tbm-polys may result in a greater $N_m$, so it is difficult to develop tight bounds for $N_m$. Obviously, the time complexity of this algorithm is directly related to $N_m$. When Poly-Merge1 is called, if we use CList-QMerge within each Poly-Merge1, the complexity is $O(N_m \times U)$. If CList-
GMerge is used within each Poly-Merge1, the complexity is $O(N_m \times U^2)$. The space used is almost the same as what is used when ImageManager→IDENTIFY-POLYS was done.

```
Algorithm ImageManager→MERGE-POLYS()
// tbm_plists is a vect of PolygonList instances, containing all the
// to-be-merged polygons.
for (i = 1; i <= threshold; i++) {
    while (tbm_plists[i]→Head-Get() is not empty) {
        poly = tbm_plists[i]→Head-Get();
        if (poly is dead because of a merge operation)
           ▷ do nothing;
        else {
            target_neibr = rule_set→Find-Target(poly);
            case_neibrs = target_neibr→Find-CascNeibrs();
            ▷ merge target_neibr, poly and case_neibrs
                and put the result polygon into result_poly;
            if (result_poly is originally a stay polygon)
               ▷ do nothing;
            else {
                if (size[result_poly] > threshold) // become a stay polygon
                    ▷ move it to stay_plist;
                else {
                    k = size[result_poly];
                    t bm_plists[k]→Tail-Add(result_poly);
                    // resort result_poly in the list of to-be-merged polygons.
                }
            }
        }
    }
} // end of while
} // end of for
```

Figure 5.17 Algorithm ImageManager→MERGE

5.4 The Algorithm for Building New Grid

Algorithm ImageManager→BUILD-NEW-IMAGE, as shown in Figure 5.18, implements the operation ImageManager→Build-New-Grid. After the algorithm ImageManager→MERGE-POLYS is completed, all to-be-merged polygons are merged and all the active polygons for the new grid are maintained on stay_plist, from which this algorithm generates the new result image. out_array is used as a work array. The column width of this array, col_max[out_array], is equal to col_max[old_image]. The great advantage of this algorithm is that row_max[out_array] can range
from 1 to row_max[old_image]. Thus, a trade-off between memory and speed can be flexibly
made. n1 and n2 record the range in the result grid that is currently processed. When rows in the
new grid from n1 to n2 are produced, n1 and n2 are updated to move to the next range, until n2
reaches the last row of the grid. For each range of n1 to n2, each polygon on stay_plist is checked.
If it has rows between the range, these rows are output. When a polygon is completely output, it is
deleted from stay_plist; otherwise it is added to the end of the PolygonList object tmp_plist. After
every polygon is processed for a range, polygons on tmp_plist are added to the end of stay_plist
and tmp_plist is cleared. This output process continues until stay_plist is n2 =
row_max[old_image].

```
Algorithm ImageManager→BUILD-NEW-GRID()
\ // out_array is used as a work array for outputing.
\ // col_max[out_array] = col_max[old_image];
\ // row_max[out_array] can range from 1 to row_max[old_image].

new_image = new Image(old_image);
tmp_plist = new PolygonList;
col_max = col_max[old_image];
row_max = row_max[old_image];
NROWS = row_max[out_array];
n1 = 1;
n2 = row_max[out_array];

while (n2 <= row_max) {
    while (stay_plist is not empty) {
        poly = stay_plist→Head_Get();
        if (size[poly] > 0) {
            poly→OUTPUT(out_array, done, n1, n2);
            if (done != true)
                tmp_plist→Tail-Add(poly);
        } else
            delete poly from stay_plist;
    }
    new_image→Write-Data(out_array, nrows);
    if (n2 == row_max) // done, every polygon is output
        return;
    n1 = n2 + 1;
    if (n2 + NROWS <= row_max)
        n2 = n2 + NROWS;
    else
        n2 = row_max;
    stay_plist→Merge-In(tmp_plist);
    tmp_plist = new PolygonList;
}
```

Figure 5.18 Algorithm ImageManager→BUILD-NEW-GRID
**Complexity analysis.** Since each cell is written out exactly once and the time complexity of finding each cell is $O(1)$, the time complexity of this algorithm is $O(n^2)$. The maximum space used is what the polygons on `stay_plist` occupied plus space of the work array `out_array` with flexible maximum row number `row_max[out_array]`. Also note that as a polygon is being output, memory for the cells which are already output is released. When the work array has only a few rows, the output process accelerates because more and more memory is released. When limited memory becomes a big issue, we can use only one row for the work array. Thus, in practice this algorithm is a dramatic improvement over a simple approach that allocates space for the entire new result grid, i.e., with best case $O(n^2)$ space costs.
6 Summary

Starting with an effort to improve the Ada program MERGE9, which implements a particular form of aggregation process, the research presented here is a comprehensive study on the nature of aggregation problems, resulting in a more formal characterization of the problem with both theoretical and practical significance. Several research efforts relating to aggregation problems are also analyzed and evaluated. A formal and strict specification is given, which includes the specification of polygon identification on grid, dissolution and merge processes, aggregation processes based on rule sets, and criteria which can be used to compare different implementations of the same rule set. The MIMRP model is an object-oriented design model introduced as an additional discipline level between the specification and algorithm design. The model permits a variety of rule set implementations, illustrated here with these specific programs. These three programs are compared with the original MERGE9 implementation, both in qualitative and quantitative terms.

This research establishes a base for a better understanding of an application that combines computer science, image processing and use of geographic information systems. Further studies can be done based on this research. For example, new criteria for optimal merge process can be introduced. The problems of, for a given set of merge rules, how to find an optimal rule implementation, and for a given rule implementation, how to find efficient algorithms, can be more efficiently addressed. The MIMRP model can also be adapted to other similar applications, and used to identify further improvements to existing implementations.
7 References


