



5-1998

## Feature extraction and retrieval using DICE : Dendronic Image Characterization Environment

Luojian Chen

Follow this and additional works at: [https://trace.tennessee.edu/utk\\_gradthes](https://trace.tennessee.edu/utk_gradthes)

---

### Recommended Citation

Chen, Luojian, "Feature extraction and retrieval using DICE : Dendronic Image Characterization Environment. " Master's Thesis, University of Tennessee, 1998.  
[https://trace.tennessee.edu/utk\\_gradthes/10177](https://trace.tennessee.edu/utk_gradthes/10177)

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact [trace@utk.edu](mailto:trace@utk.edu).

To the Graduate Council:

I am submitting herewith a thesis written by Luojian Chen entitled "Feature extraction and retrieval using DICE : Dendronic Image Characterization Environment." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael W. Berry, Major Professor

We have read this thesis and recommend its acceptance:

Jens Gregor, Brad Vander Zanden

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Luo Jian Chen entitled "Feature Extraction and Retrieval Using DICE: Dendronic Image Characterization Environment." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

*Michael W. Berry*

Dr. Michael W. Berry, Major Professor

We have read this thesis  
and recommend its acceptance:

*Brad Vander Zander*

*Ken Guey*

Accepted for the Council:

*Lew Minkal*

Associate Vice Chancellor  
and Dean of the Graduate School

**Feature Extraction and Retrieval Using  
DICE: Dendronic Image  
Characterization Environment**

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Luojian Chen

May 1998

## Acknowledgments

I thank my advisor, Dr. Michael Berry, and Dr. William Hargrove for their patience, encouragement, support, and advice throughout this project. I would also like to thank Drs. Jens Gregor and Brad Vander Zanden for serving on my thesis committee. Most importantly, I am grateful to my parents for supporting my study abroad.

This research has been supported by the National Science Foundation under Grant Nos. ASC-94-11394 and CDA-95-29459.

## Abstract

Due to the enormous growth of digital image collections, image analysis and retrieval is of vital importance. Traditional image analysis and retrieval techniques may be time consuming and inaccurate. This thesis presents a new technique for image analysis and retrieval based on dendronic image signatures. Dendronic image characterization, as implemented in the **DICE (Dendronic Image Characterization Environment)** software environment, is a data-driven and self-structuring process, which can be used in many application areas such as military surveillance, medical image analysis, and computer graphics. The algorithm for dendrone construction is robust and the algorithm for matching sub-dendrones (i.e., feature extraction) according to distance-to-centroid signatures is simple and effective. The DICE software environment presented in this thesis is flexible, extendable, and portable.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation and Goals . . . . .	1
1.2	Overview . . . . .	3
<b>2</b>	<b>Overview of Dendrone Data Structure and Algorithms</b>	<b>4</b>
2.1	Dendrone Data Structure . . . . .	4
2.1.1	Construction of Dendrones . . . . .	4
2.1.2	Properties of Dendrones . . . . .	13
2.1.3	Uses of Dendrones . . . . .	16
2.2	Algorithms . . . . .	16
2.2.1	Dendrone Construction Algorithms . . . . .	17
2.2.2	Sub-Dendrone Matching Algorithms . . . . .	20
<b>3</b>	<b>The Design and Implementation of DICE</b>	<b>36</b>
3.1	The Dendrone Library . . . . .	37
3.1.1	The Image Class . . . . .	40

3.1.2	The XpmImage Class . . . . .	40
3.1.3	The Histogram Class . . . . .	40
3.1.4	The Option Class . . . . .	40
3.1.5	The Object Class . . . . .	41
3.1.6	The Dendrone Class . . . . .	41
3.1.7	The Shape Class . . . . .	51
3.1.8	The RankList Class . . . . .	51
3.2	The Graphical User Interface . . . . .	52
3.2.1	Generating Dendrones from Images . . . . .	53
3.2.2	Reconstructing Images from Dendrones . . . . .	55
3.2.3	Object Retrieval by Matching Sub-Dendrones . . . . .	58
3.2.4	Customizing DICE . . . . .	60
3.3	The Interface Module . . . . .	61
<b>4</b>	<b>Evaluating the Performance and Effectiveness of DICE</b>	<b>63</b>
4.1	Performance Evaluation . . . . .	63
4.1.1	Procedure . . . . .	64
4.1.2	Results . . . . .	66
4.2	Effectiveness Evaluation . . . . .	78
4.2.1	Matching Artificial Shapes . . . . .	79
4.2.2	Matching Objects from Real Images . . . . .	82
<b>5</b>	<b>Conclusions</b>	<b>85</b>



<b>Bibliography</b>	<b>87</b>
<b>Appendix API Tables of C++ Classes in DICE</b>	<b>90</b>
<b>Vita</b>	<b>101</b>

# List of Tables

3.1	File management in DICE. . . . .	52
4.1	Images used to evaluate the performance of the dendrone construction algorithms. . . . .	64
4.2	Summary of dendrone construction performance figures and tables. . . .	66
4.3	Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.1 using the pixel labeling algorithm. . . . .	68
4.4	Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.1 using the pixel labeling algorithm.	68
4.5	Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.1 using the recursive Connectivity Filling algorithm. . . . .	70
4.6	Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.1 using the recursive Connectivity Filling algorithm. . . . .	70

4.7	Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.1 using the non-recursive Connectivity Filling algorithm.	72
4.8	Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.1 using the non-recursive Connectivity Filling algorithm. . . . .	72
4.9	Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.2 using the pixel labeling algorithm. . . . .	74
4.10	Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.2 using the pixel labeling algorithm.	74
4.11	Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.2 using the non-recursive Connectivity Filling algorithm.	76
4.12	Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.2 using the non-recursive Connectivity Filling algorithm. . . . .	76
4.13	Memory usage of dendrone construction (stride = 30). . . . .	78
4.14	Summary of images used to evaluate the effectiveness of the distance-to-centroid image signature matching algorithm. . . . .	79
A.1	API methods of the <i>Image</i> class. . . . .	91
A.2	API methods of the <i>XpmImage</i> class. . . . .	91
A.3	API methods of the <i>Histogram</i> class. . . . .	92
A.4	API methods of the <i>Option</i> class (part 1). . . . .	93

A.5	API methods of the <i>Option</i> class (part 2).	94
A.6	API methods of the <i>Object</i> class (part 1).	95
A.7	API methods of the <i>Object</i> class (part 2).	96
A.8	API methods of the <i>Dendrone</i> class (part 1).	97
A.9	API methods of the <i>Dendrone</i> class (part 2).	98
A.10	API methods of the <i>Dendrone</i> class (part 3).	99
A.11	API methods of the <i>Shape</i> class.	99
A.12	API methods of the <i>RankList</i> class.	100

# List of Figures

2.1	Image showing three objects. . . . .	5
2.2	Histogram of intensity levels represented in the image from Figure 2.1. .	6
2.3	Imaginary three-dimensional intensity terrain generated from the image in Figure 2.1. . . . .	6
2.4	Dendrogram of the image in Figure 2.1. . . . .	8
2.5	Image (a) showing three separate objects appearing when the water level is 255 and its sub-dendrogram (b). . . . .	10
2.6	Image (a) showing two of the three separate objects growing larger when the water level is 165 and its sub-dendrogram (b). . . . .	11
2.7	Image (a) showing two separate objects merging and forming one larger object when the water level is 135 and its sub-dendrogram (b). . . . .	12
2.8	Enlarged and rotated version of the image in Figure 2.1. . . . .	14
2.9	Dendrogram of the image in Figure 2.8. . . . .	15
2.10	Bounding boxes of objects. . . . .	21

2.11 One object extracted (a) from the image in Figure 2.1 along with the original edge (b), thinned edge (c), and final edge (d). . . . .	25
2.12 (a) Image before m-connectivity is checked. (b) Image after m-connectivity is checked. . . . .	27
2.13 Image showing one object with four pixels labeled. . . . .	30
2.14 The edge signature of the object in Figure 2.13. Four pixels are labeled accordingly. . . . .	31
2.15 One object (rotated object from Figure 2.13) with four pixels labeled. .	33
2.16 The edge signature of the object in Figure 2.15. Four pixels are labeled accordingly. . . . .	33
2.17 Signature image matching. . . . .	34
3.1 Interactions among DICE software modules and the user. . . . .	38
3.2 Dendrone library class hierarchy. . . . .	39
3.3 Steps of the construction of a dendrone. . . . .	42
3.4 Dendrone data structure implementation. . . . .	44
3.5 <i>X</i> -coordinate dendrogram of the image in Figure 2.1. . . . .	46
3.6 <i>Y</i> -coordinate dendrogram of the image in Figure 2.1. . . . .	47
3.7 <i>DICE</i> main window. . . . .	53
3.8 <i>ImageDisplayer</i> window for dendrone construction. . . . .	54
3.9 <i>BuildDialog</i> window for dendrone construction. . . . .	54
3.10 <i>DendroneDisplayer</i> window. . . . .	55

3.11	<i>ReconstructDialog</i> window for image reconstruction. . . . .	56
3.12	<i>ImageDisplayer</i> window for image reconstruction. . . . .	57
3.13	<i>MatchDialog</i> window for object matching. . . . .	59
3.14	<i>MatchResultDisplayer</i> window showing object matching results. . . . .	60
3.15	<i>AppConfigDialog</i> window for external application program configuration. . . . .	61
3.16	Calling C++ methods in Java. . . . .	62
4.1	High-altitude photograph of a Boeing 747 in flight. . . . .	64
4.2	A biplane and its shadow as seen from above. . . . .	65
4.3	Total elapsed time of dendrone construction for the image in Figure 4.1 using the pixel labeling algorithm. . . . .	67
4.4	Total elapsed time of dendrone construction for the image in Figure 4.1 using the recursive Connectivity Filling algorithm. . . . .	69
4.5	Total elapsed time of dendrone construction for the image in Figure 4.1 using the non-recursive Connectivity Filling algorithm. . . . .	71
4.6	Total elapsed time of dendrone construction for the image in Figure 4.2 using the pixel labeling algorithm. . . . .	73
4.7	Total elapsed time of dendrone construction for the image in Figure 4.2 using the non-recursive Connectivity Filling algorithm. . . . .	75
4.8	Images used to evaluate the effectiveness of the distance-to-centroid image signature matching algorithm. . . . .	80
4.9	Matching shapes from Figure 4.8(b). . . . .	81

4.10 Matching objects from Figure 4.8(d) . . . . . 84



# Chapter 1

## Introduction

Images are being generated at an increasing rate by sources such as defense and civilian satellites, military reconnaissance, and biomedical imaging (see [GR95]). New image analysis and retrieval techniques are required to effectively extract and use information from these images.

### 1.1 Motivation and Goals

As described in [GR95], among many different ways, previous approaches to image analysis and content-based retrieval have mainly taken two directions. The first approach models image contents as a set of attributes extracted manually and managed within conventional database-management systems, which entails a high level of image abstraction. For example, the *Chabot* system (see [OS95]) developed at UC Berkeley uses a relational database to store text information describing certain attributes of im-

ages (photos), which are searched when a user enters a query. These attributes include abstract, title, comments, copyright information, location where the photo is taken and date when the photo is taken.

The second approach uses an integrated feature-extraction/object-recognition subsystem to overcome the limitations of attribute-based retrieval. However, this approach is often computationally expensive, difficult, and tends to be domain-specific. One example of this approach is the *QBIC* (Query by Image Content) system (see [FSN<sup>+</sup>95]) developed at IBM Almaden Research Center, which allows queries on large image and video databases based on example images, user-constructed sketches and drawings, selected color and texture patterns, camera and object motion, and other graphical information.

This thesis presents a new technique for image analysis and retrieval based on dendronic image signatures. A **dendrone** is a searchable hierarchical thresholding structure generated from an image. The dendrone structure captures the unique signatures of objects within the image and sub-dendrones within the hierarchy are recognizable as objects within the image. A dendrone can be used as a framework for image analysis and retrieval. While the dendrone itself does not impose external restrictions to the image, certain attributes could be incorporated into the dendrone. For example, text information describing the image could be added to the dendrone to facilitate content-based retrieval. Different feature description attributes could also be computed and stored in the dendrone for shape-based object retrieval. An implementation of the dendrone

technique, the **DICE** (**Dendronic Image Characterization Environment**) software environment is presented in this thesis. DICE provides an integrated tool for image feature extraction and object retrieval based on dendronic image signatures.

## 1.2 Overview

The following chapters discuss the theory, implementation, and evaluation of dendronic image characterization. Chapter 2 reviews the underlying concepts of dendrones and discusses the algorithms for dendrone construction and sub-dendrone matching. Chapter 3 introduces the design and implementation of the DICE software environment. Chapter 4 examines the efficiency of dendrone construction and effectiveness of sub-dendrone matching. Finally, Chapter 5 summarizes the usefulness and effectiveness of dendronic image characterization.

## Chapter 2

# Overview of Dendrone Data Structure and Algorithms

### 2.1 Dendrone Data Structure

This section presents the searchable dendrone [Har97] data structure generated from raw unsearchable digital images, the algorithms used to construct dendrones and match sub-dendrones.

#### 2.1.1 Construction of Dendrones

An image may be considered as a two-dimensional intensity field. In addition, if the intensity values of each pixel in the image is considered as elevations, an image can be viewed as a three-dimensional intensity terrain. The brighter the pixel is, the higher the elevation. Brighter pixels form *mountains* with the brightest pixel as the peak. Darker

pixels form *valleys* with the darkest pixel as the bottom. However, this imaginary terrain differs from real-world geographical terrain in that the imaginary terrain is *solid*, which means it has no subsurface features such as *holes* or *caves*.

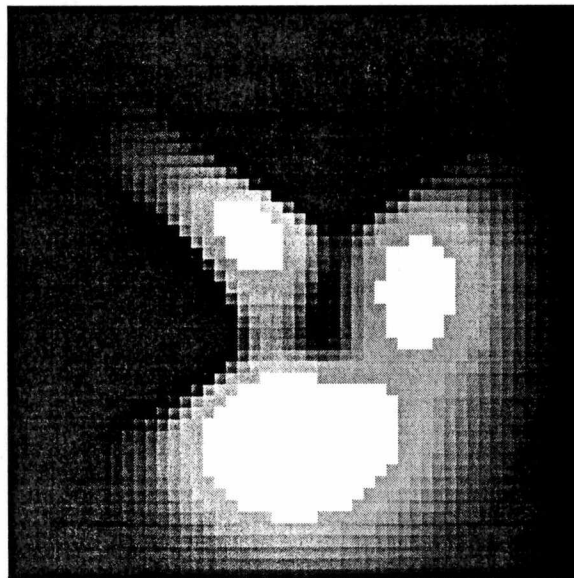


Figure 2.1: Image showing three objects.

Figure 2.1 is an artificially generated greyscale image showing three objects. Figure 2.2 is the histogram of intensity levels represented in the image from Figure 2.1. The distribution of different intensity levels ranging from 0 to 255 are represented in this graph. Figure 2.3 is the three-dimensional intensity terrain corresponding to the image in Figure 2.1. In the original image, the brightest intensity value is 255 and the darkest intensity value is 0. Accordingly, in the imaginary three-dimensional intensity terrain, the highest elevation is 255 and lowest elevation is 0.

Given an image, a unique dendrone structure can be constructed from the imaginary

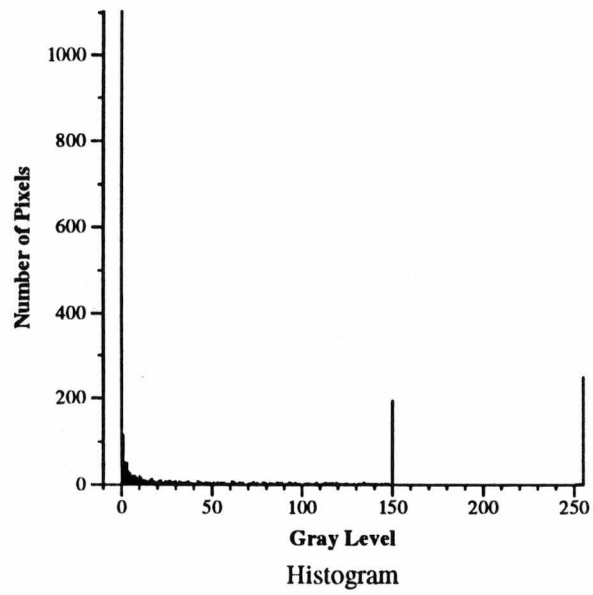


Figure 2.2: Histogram of intensity levels represented in the image from Figure 2.1.

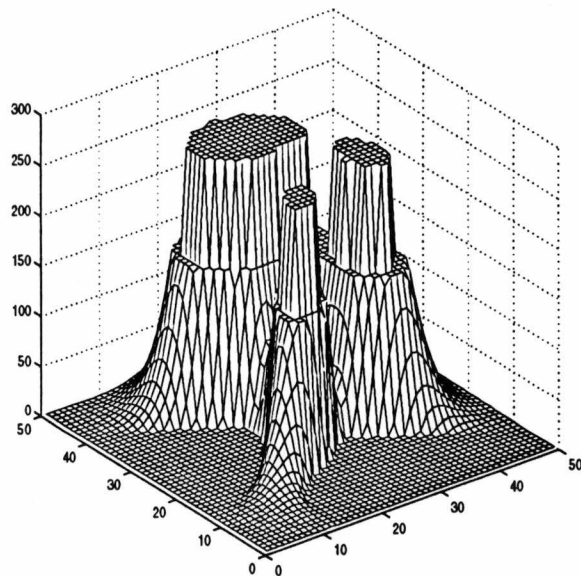


Figure 2.3: Imaginary three-dimensional intensity terrain generated from the image in Figure 2.1.

three-dimensional terrain. The dendrone structure captures the connectedness of objects and sub-objects during successive brightness thresholding. The construction process can be visualized as if the terrain is first flooded with water and then the water is slowly drained away. Initially, the water level is so high that no *land* is visible above the water level. As the water level decreases, *mountains* associated with the higher elevations appear first, then *plains*, and finally *valleys*. At any particular water (intensity) level, the image is segmented into *islands* (objects). When the water (intensity) level decreases, only three kinds of events occur:

1. New isolated islands (objects) appear above the water level.
2. Existing islands (objects) grow in size.
3. Multiple islands (objects) merge or coalesce and form larger islands (objects).

Figure 2.4 illustrates the tree-like **dendrogram** corresponding to the dendrone generated from Figure 2.1. This figure illustrates one of many possible ways to draw the dendrone graphically. The dendrone, in this case, is generated with the intensity level decreasing at an intensity resolution increment (stride value) of 30. The smaller the stride value, the more detailed the dendrone in terms of the number and size of subtrees. It is clear that there are three distinct sub-dendrones within the dendrone (see Figure 2.4), which correspond to the three distinct objects in Figure 2.1. In this dendrogram, the horizontal axis is arbitrary and the vertical axis indicates the intensity value from 0 to 255. Each vertical line of the dendrogram corresponds to one object

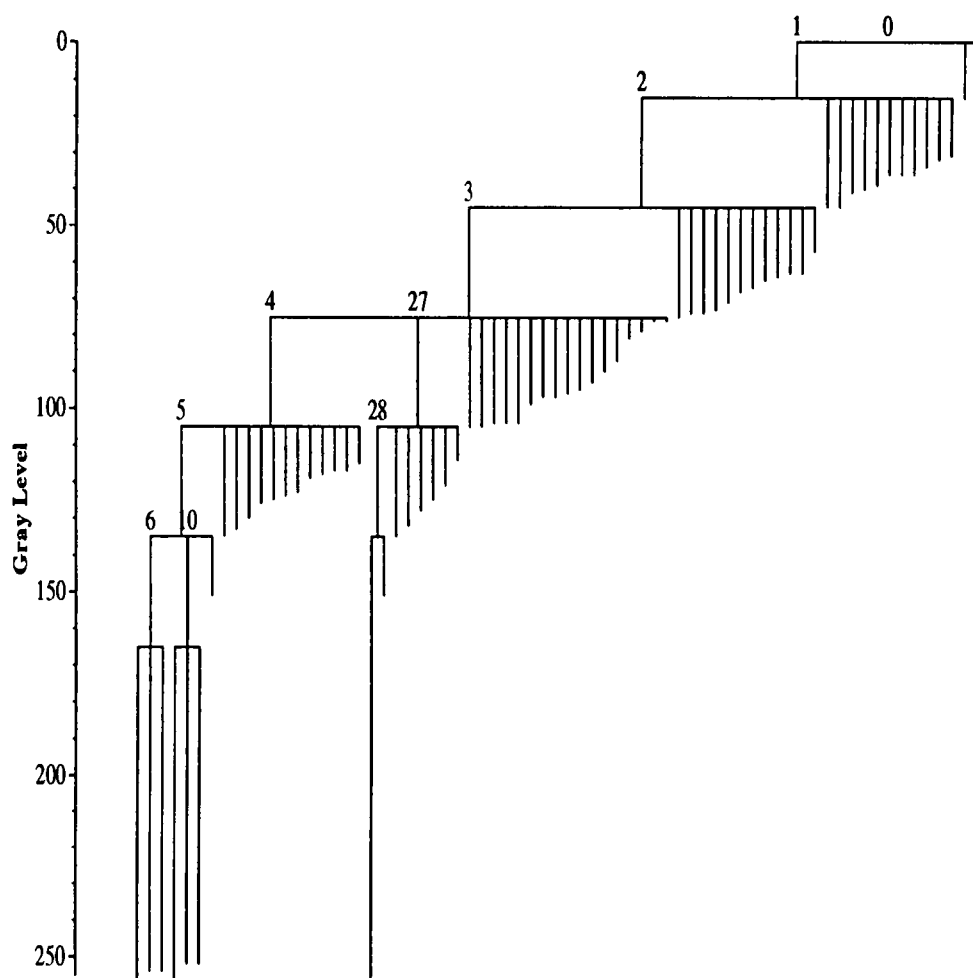
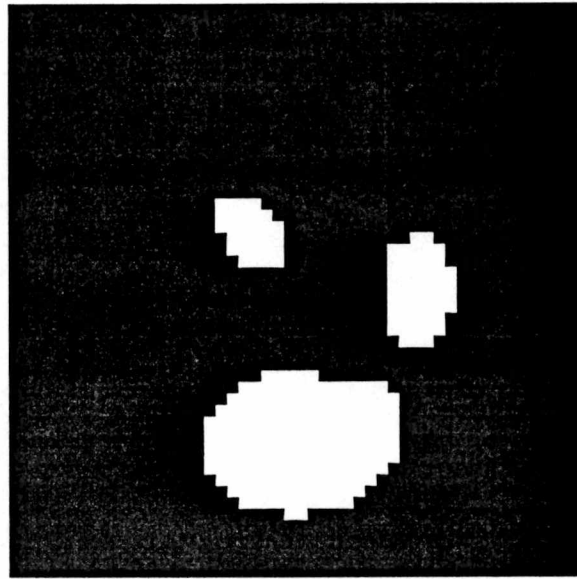


Figure 2.4: Dendrogram of the image in Figure 2.1.

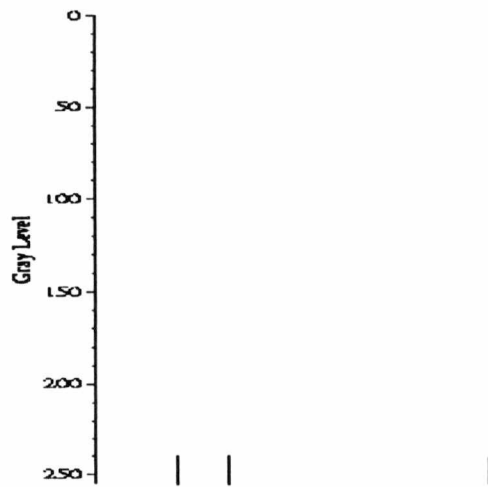


within the image. The length of the line indicates the intensity of the object. Each horizontal line connects one vertical line above with several other vertical lines below so that the object (actually the parent object identified by the number above the horizontal line) represented by the vertical line above the horizontal line is formed by those objects (the sub-objects or child objects) represented by the vertical lines beneath the horizontal line. The position of the horizontal lines indicates the intensity level at which the image is segmented. The intensity level decreases from 255 to 0 at a specified stride value. When the intensity level reaches 0, the entire image forms one object, which is represented by the root or the trunk of the dendrone. Despite the flooding/draining allegory, the dendrogram cannot be drawn or constructed until all of the water has been drained and the thresholding is complete. Until then, it is unknown where on the arbitrary horizontal axis the vertical lines should be drawn so that they can be connected properly. It is the connectedness as one traverses the dendrone tree from root to leaves that contains the information about the relationships between objects, which can be used to reconstruct part or all of the original image.

Figures 2.5, 2.6, and 2.7 are images and dendrograms generated from the original image in Figure 2.1 when the water level is three different values. Figure 2.5(a) shows that when the water level starts at 255, three separate objects begin to appear. Correspondingly, in Figure 2.5(b), there are three vertical lines representing these three objects. Since they are distinct objects, the three vertical lines have no connections among them. As the water level decreases to 165, two of the three objects have grown

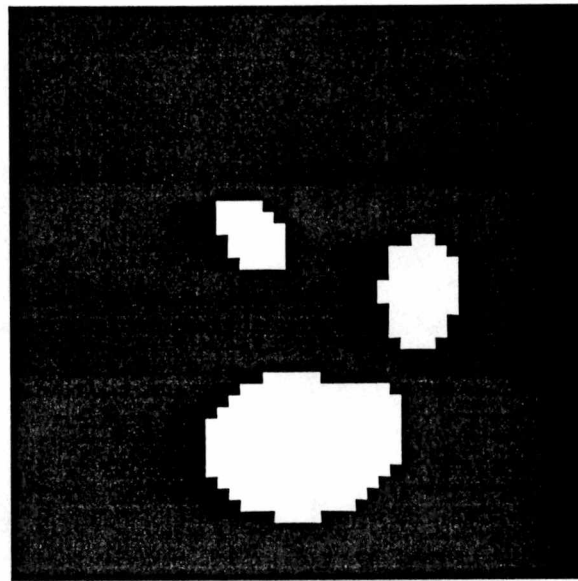


(a)

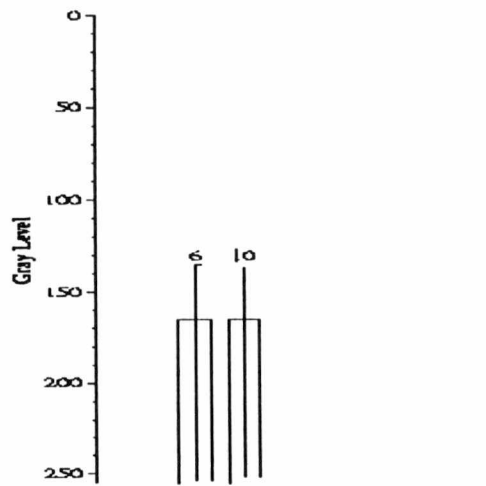


(b)

Figure 2.5: Image (a) showing three separate objects appearing when the water level is 255 and its sub-dendrogram (b).

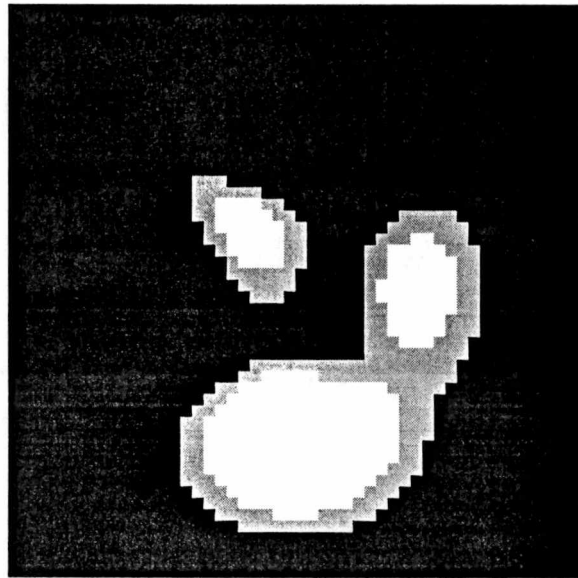


(a)

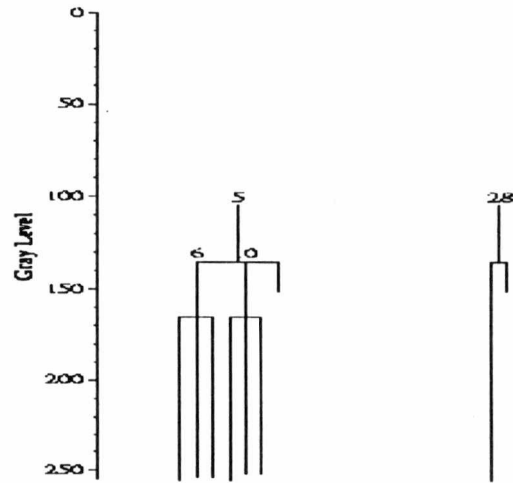


(b)

Figure 2.6: Image (a) showing two of the three separate objects growing larger when the water level is 165 and its sub-dendrogram (b).



(a)



(b)

Figure 2.7: Image (a) showing two separate objects merging and forming one larger object when the water level is 135 and its sub-dendrogram (b).

larger (see Figure 2.6(a)) as more components appear. In Figure 2.6(b), the larger objects are represented by the vertical lines identified by integers *6* and *10*. They link their corresponding sub-objects together by the horizontal lines whose *y*-coordinate positions are 165. The other object's size does not increase at this moment. Figure 2.7(a) shows that when the water level is 135, which is below the elevation of a connected portion shared by two of the three objects, these two objects merge and form a larger object, which is identified by the integer *5* in the dendrogram in Figure 2.7(b). At that water level, the third object also grows and becomes a larger object, which is identified by integer *28* in the dendrogram. In real-world (more complex) images, the segmentation and merging processes are much more complicated and the three events described above may simultaneously occur and involve more than two objects.

### 2.1.2 Properties of Dendrones

Dendrones generated from images are invariant to scaling. The information stored in the nodes in the dendrones may be different, but the overall structure of the dendrones does not change. Dendrones are also invariant to rotation from a connectedness standpoint although they are not necessarily structurally equivalent. In other words, the relationships among child objects and parent objects will not change although the order of objects within the dendrones may be different. Figure 2.8 illustrates an enlarged and rotated version of the image in Figure 2.1. The dendrogram generated from this image (see Figure 2.9) is exactly the same as the dendrogram shown in Figure 2.4. Dendrones are invariant (from a connectedness standpoint) to the placement of objects within the

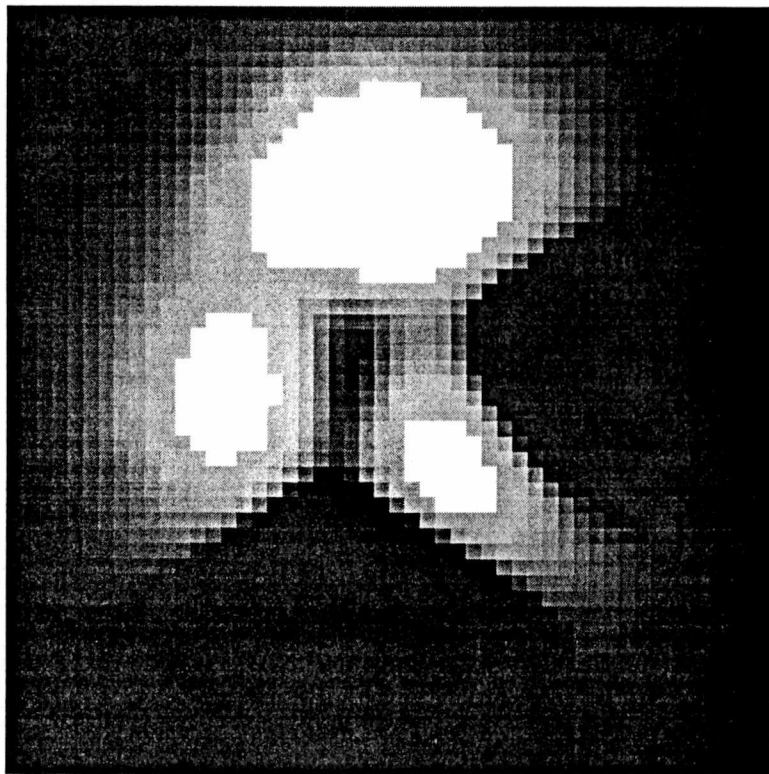


Figure 2.8: Enlarged and rotated version of the image in Figure 2.1.

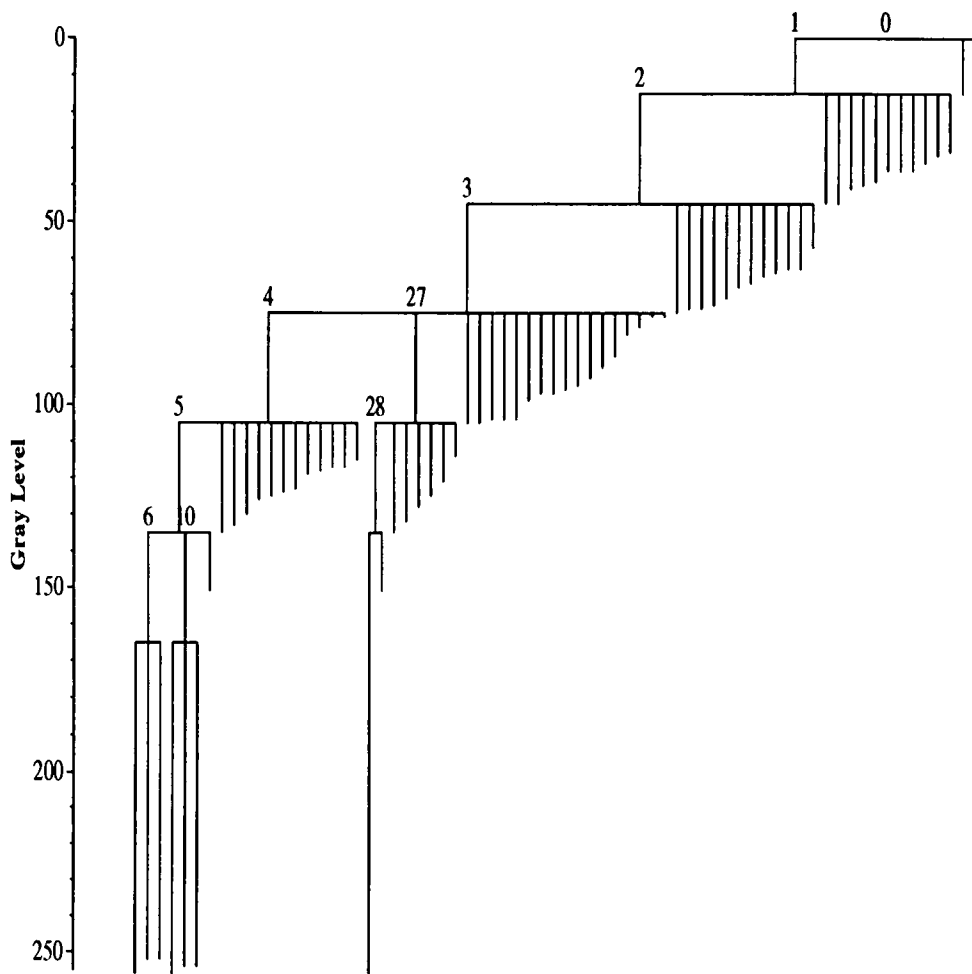


Figure 2.9: Dendrogram of the image in Figure 2.8.

image and subsequent intensity changes as long as the relative intensity relationships among the objects remain the same.

### **2.1.3 Uses of Dendrones**

The dendrone data structure provides a useful and powerful computational framework for image analysis and visual information retrieval. Each node (branching point) of the dendrone can store information such as the position of the object within the image, the size of the object, its eccentricity and axis orientation and information that is directly available from the image itself (e.g., the intensity values of the pixels).

If the dendrone stores information such as pixel coordinates and intensity values, the original image can be reconstructed from the dendrone. Furthermore, one can detect individual objects from the image by extracting sub-dendrones from the dendrone. In **DICE** (**Dendronic Image Characterization Environment**) (see Chapter 3), dendrones are used to detect objects with similar shape and/or similar surface structure from multiple images.

## **2.2 Algorithms**

In **DICE**, fast algorithms are used to generate dendrones and match sub-dendrones. The construction of dendrones involves segmenting the image into isolated objects and building the tree structure from these objects. The matching of dendrones is more complicated than the construction of dendrones. As mentioned above, in **DICE**, the



primary goal is to match dendrones representing objects with similarity shape and/or surface structure from multiple images. The structure of the dendrone is sufficient to match objects for similar surface structure. However, matching objects for similar shape is more difficult. Many algorithms and techniques have been developed in image analysis for shape matching. In DICE, a simple but effective **distance-to-centroid** [Rau94] signature matching algorithm is used. Of course, this particular method is not the only way or necessarily *best* way to match objects from random images. DICE is designed to be flexible enough to allow different matching algorithms to be used (see Chapter 3).

### 2.2.1 Dendrone Construction Algorithms

The construction of dendrones is accomplished by thresholding the image in a repetitive fashion. At one particular threshold intensity level, the image is processed in two stages:

1. Image segmentation.
2. Object merging.

#### Image Segmentation Algorithm

The pixel labeling algorithm presented in [Jai89] can be used to segment the image into isolated objects. The image is scanned from left to right and top to bottom. The

current pixel is labeled according to its intensity value. Consider the collection of pixels

$$\begin{pmatrix} A & B & C \\ D & X & \end{pmatrix},$$

where the current pixel is  $X$ . The pixels above and to left of the current pixel have already been labeled if they are within the current threshold range. If the intensity value of the current pixel is within the current threshold range, pixels  $A$ ,  $B$ ,  $C$ , and  $D$  are examined and one of the following situations can occur:

1. None of these pixels are labeled.

Pixel  $X$  is assigned a new label. A new object is created with one pixel  $X$  in it.

2. One of these pixels is labeled.

Its label is assigned to  $X$ . For example, if  $C$  is the only pixel that has been labeled,  $C$ 's label is assigned to  $X$  and pixel  $X$  is added to the object with that label.

3. There are two or more qualified labels.

These labels are declared to be the same and updated with a new label which is assigned to  $X$ . The objects associated with these labels are merged to become a new object (with the new label) and pixel  $X$  is added to the new object.

Other existing image segmentation algorithms include Connectivity Filling [Pav82], Amplitude Thresholding or Window Slicing [Jai89], and Run-length Connectivity Analysis [Jai89]. Detailed discussion of the performance of the pixel labeling algorithm, the

recursive version of Connectivity Filling, and the non-recursive version of Connectivity Filling is presented in Chapter 4.

### **Object Merging Algorithm**

After all pixels have been scanned, the image is segmented into isolated objects, each with a distinct label. These objects are then examined along with any objects generated from previous iterations of the segmentation and merging processes.

The object merging algorithm compares each object generated from the current segmentation process with any object previously generated. If two objects are connected or *touch*, they are merged. Newly generated objects are stored together and will be examined when the image is segmented at a lower threshold intensity level. The newly generated objects also link their respective sub-objects together. If an object does not touch any other object, it is also stored with other newly generated objects and will be examined in future segmentations. The steps of detecting whether two objects, object *A* and object *B*, touch each other are:

1. If object *A* is a composite object (object that has sub-objects), check each of its sub-objects against object *B*. If any of these sub-objects touches object *B*, object *A* touches object *B*.
2. If object *B* is a composite object, check object *A* against each of object *B*'s sub-objects. If object *A* touches any of these sub-objects, object *A* touches object *B*.

3. If both object *A* and object *B* are primitive objects (objects that have no sub-objects):
  - (a) If the bounding boxes of the two objects do not overlap, object *A* does not touch object *B*. The bounding box of an object is the smallest rectangle containing the object. In addition, the edges of the box must be parallel to the axes of the coordinate system. In Figure 2.10, object 1's bounding box and object 2's bounding box do not overlap. Therefore, these two objects do not touch.
  - (b) Otherwise, get the intersecting (overlapping) part of the bounding boxes. Extract pixels from both objects that are inside the intersecting part. In Figure 2.10, object 3's bounding box and object 4's bounding box overlap. The intersecting part is the shaded area.
  - (c) Check the pixels obtained from the previous step. If there is a pixel from object *A* that is adjacent to a pixel from object *B*, object *A* touches object *B*. In Figure 2.10, although object 3's bounding box and object 4's bounding box overlap, no pixels are adjacent. Therefore, these two objects do not touch.

### 2.2.2 Sub-Dendrone Matching Algorithms

In DICE, there are two types of sub-dendrone matching:

1. Matching based on the structures of the dendrones.

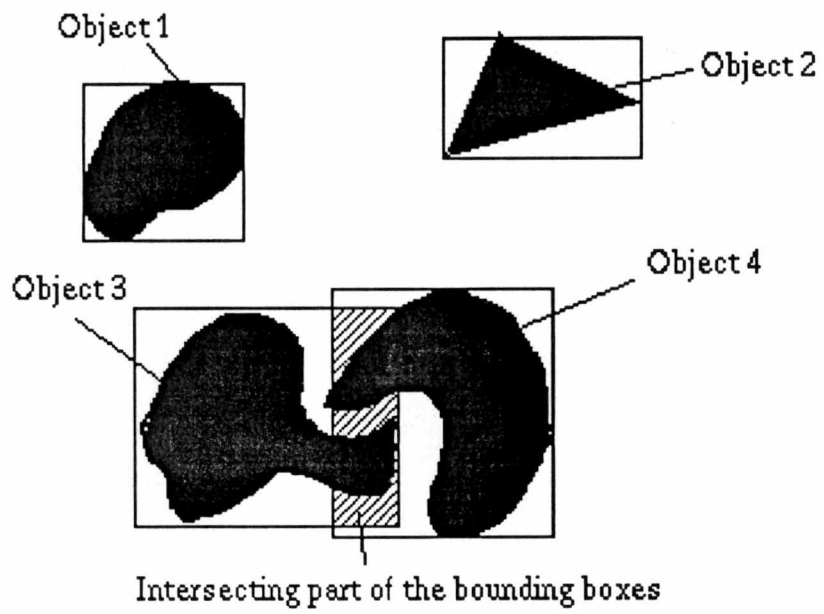


Figure 2.10: Bounding boxes of objects.

## 2. Matching based on the shapes of the objects represented by the dendrones.

As mentioned previously in Section 2.2, there are potentially many different ways to compare sub-objects. In DICE, just a few candidate methods based on sub-dendrone matching are currently used.

### Structure-Based Matching

The structure-based matching algorithm takes two dendrones as inputs. One is the *source dendrone* computed from the source image. Usually it is a sub-dendrone extracted from a complete dendrone. The other is the *target dendrone*, which is usually computed from a complete image.

The structure-based matching algorithm does a breadth-first search on the target dendrone. At every node, the algorithm computes the difference between the number of descendent nodes in the source dendrone and the number of descendent nodes in the target dendrone. For each descendent node, it recursively computes the difference also. A similarity value between 0 and 1 is assigned to every child node to indicate how similar two sub-dendrones are. A similarity value of 1 indicates the two sub-dendrones are identical and a similarity value of 0 means the two sub-dendrones have no similarities.

Several simple rules have been applied to calculate the similarity between two sub-dendrones:

1. Leaf sub-dendrone (dendrones that have only one node) and non-leaf sub-dendrone (dendrones that have at least two nodes) have a similarity value of 0.

2. Two leaf sub-dendrones (dendrones that have only one node) have a similarity value of 1.
3. The similarity value of two non-leaf dendrones is computed as:

$$S_{ij} = \frac{\sum \left( S_{kl} \times \frac{N_k + N_l}{2} \right)}{\sum \left( \frac{N_k + N_l}{2} \right) + \sum N_m}, \quad (2.1)$$

where

$S_{ij}$  is the similarity between dendrone  $i$  and dendrone  $j$  and the total number of objects in dendrone  $i$  is not more than the total number of objects in dendrone  $j$ ,

$S_{kl}$  is the best-fit similarity between sub-dendrone  $k$ , which is a sub-dendrone of dendrone  $i$ , and sub-dendrone  $l$ , which is a sub-dendrone of dendrone  $j$ ,

$N_k$  is the total number of objects in sub-dendrone  $k$ ,

$N_l$  is the total number of objects in sub-dendrone  $l$ , and

$N_m$  is the total number of objects in sub-dendrone  $m$ , which is an unmatched sub-dendrone in dendrone  $j$ .

Finally, the weighted average value  $S_{ij}$  (see Equation (2.1)) is used to represent the overall similarity between the source dendrone and every sub-dendrone within the target dendrone.

## Shape-Based Matching

Shape is an important feature to identify or match objects from multiple images. There are many techniques in shape representation and matching. Some examples are boundary representation techniques [Jai89] (including Chain Codes, Fitting Line Segments, B-Spline Representation, Control Points, and Fourier Descriptors), Region Representations [Jai89] (including Run-length Codes, Quad-trees, and Projections), and Moment Representations [Jai89]. In DICE, a distance-to-centroid representation is used.

Similar to the structure-based matching algorithm, the shape-based matching algorithm does a breadth-first search on the target dendrone. At every node, the distance-to-centroid signature of the object represented by that sub-dendrone is computed. The signature is then compared with the signature of the object represented by the source dendrone.

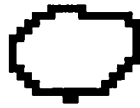
The procedure to compute the distance-to-centroid signature of an object is as follows:

1. The pixels of the object are extracted from the dendrone and an image of the object is reconstructed. In the original image, these pixels may have different intensity values, but in the reconstructed image they all have the same intensity value. Figure 2.11(a) shows the image of one object extracted from the image in Figure 2.1.
2. The edge pixels are extracted and an image of the object's edge is obtained. An edge pixel is defined to be a pixel whose eight-neighbor pixels are not all pixels of

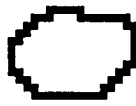




(a)



(b)



(c)



(d)

Figure 2.11: One object extracted (a) from the image in Figure 2.1 along with the original edge (b), thinned edge (c), and final edge (d).

the object. Figure 2.11(b) shows the edge image of the object from Figure 2.11(a).

3. A thinning algorithm [ZS84] is applied to the edge image so that the edges are at most one pixel wide. After the thinning algorithm is applied, a skeleton image of the object's edge is obtained. Figure 2.11(c) shows the thinned edge image of the object from Figure 2.11(a).
4. Each edge pixel is examined for  $m$ -connectivity [GW87]. Pixels  $p$  and  $q$  are said to be  $m$ -connected if one of the two following situations occurs:

- (a)  $q \in N4(p)$ , where  $N4(p)$  is the set of four neighboring pixels of pixel  $p$ . For example, in the collection of pixels

$$\begin{pmatrix} & q & \\ X & p & X \\ & X & \end{pmatrix},$$

where pixels labeled  $X$ 's are the pixels in set  $N4(p)$ , pixels  $p$  and  $q$  are  $m$ -connected.

- (b)  $q \in Nd(p)$ , where  $Nd(p)$  is the set of four diagonal neighbor pixels of pixel  $p$ , and  $N4(p) \cap N2(q) \neq \emptyset$ , where  $N2$  is the set of two neighboring pixels of pixel  $q$ . For example, in the collection of pixels

$$\begin{pmatrix} X & Y & q \\ & p & Y \\ X & & X \end{pmatrix},$$

where pixels labeled  $X$ 's are the pixels in set  $Nd(p)$  and pixels labeled  $Y$ 's are the pixels in set  $N2(q)$ , pixels  $p$  and  $q$  are  $m$ -connected.

Some pixels are deleted to guarantee that there is only one path between any two pixels in the edge image and ensure that there is no ambiguity with respect to edge traversal. Figure 2.11(d) shows the final edge image of the object from Figure 2.11(a).

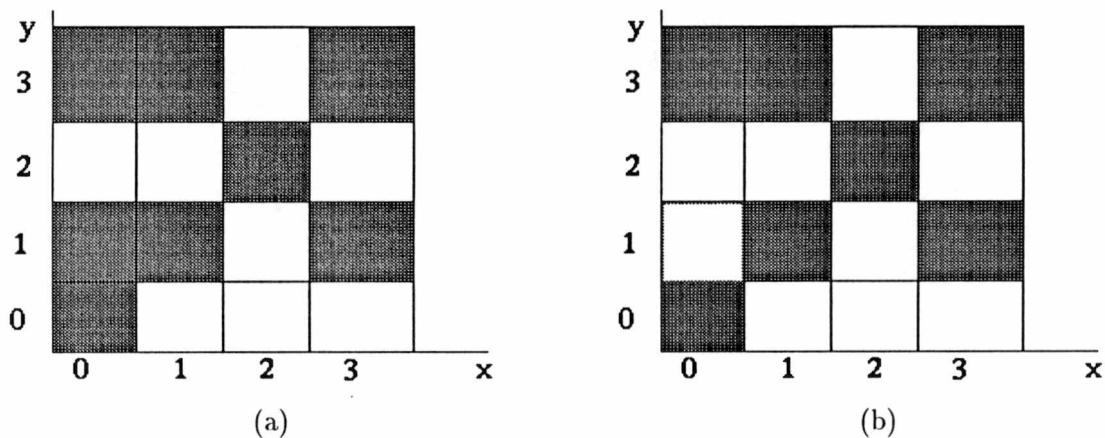


Figure 2.12: (a) Image before  $m$ -connectivity is checked. (b) Image after  $m$ -connectivity is checked.

Figure 2.12(a) illustrates an image before  $m$ -connectivity is checked. In the image, there are two possible paths from pixel  $(0, 0)$  to pixel  $(1, 1)$ . One is from pixel  $(0, 0)$  to pixel  $(1, 1)$  directly and the other one is via pixel  $(0, 1)$ . A mask is passed over the image which searches for certain patterns and eventually deletes the pixel that causes the ambiguity, in this case, pixel  $(0, 1)$  (see Figure 2.12(b)).

5. The centroid (the geometrical center) of the object and the distances from all edge

pixels to the centroid are computed. The edges are interpolated in order to make the signature invariant to scaling. Since the image is digital, the objects in the image are discrete. Consequently, the edge curves of the objects are also discrete. One simple approach to approximate a discrete curve pattern is to compose it by all line segments of neighboring pixels. As presented in [Rau94], an *analytical* definition of a *discrete* line segment between two points  $z_i = (x_i, y_i)$  and  $z_j = (x_j, y_j)$  is given by

$$z_{ij}(t) = (x_i + t(x_j - x_i), y_i + t(y_j - y_i)), t \in (0, 1).$$

A discrete object  $\hat{\Omega}(t)$  consisting of  $n$  pixels  $z_i = (x_i, y_i)$  is then composed of the union of  $m$  line segments  $z_{ij}(t)$  between all neighboring pixels  $z_i = (x_i, y_i)$  and  $z_j = (x_j, y_j)$  of the object [Rau94]. Specifically,

$$\hat{\Omega}(t) = \bigcup z_{ij}(t), z_{ij}(t) = (x_i + t(x_j - x_i), y_i + t(y_j - y_i)), t \in (0, 1).$$

The centroid  $O$  of an object is calculated as

$$O = (O_x, O_y), O_x = \frac{\sum_{i=1}^n x_i}{n}, O_y = \frac{\sum_{i=1}^n y_i}{n},$$

where

$O_x$  is the  $x$ -coordinate of the centroid,

$O_y$  is the  $y$ -coordinate of the centroid,

$x_i$  is the  $x$ -coordinate of one edge pixel,

$y_i$  is the  $y$ -coordinate of one edge pixel, and

$n$  is the total number of edge pixels.

When computing the distances from edge pixels to the centroid, the maximum distance,  $M$ , from the edge pixels to the centroid is also calculated. The maximum Euclidean distance  $M$  of any pixel to the centroid is calculated as

$$M = \max \left( \sqrt{(x_i - O_x)^2 + (y_i - O_y)^2} \right), i = 1, 2, \dots, n.$$

In the signature image, the distances of all pixels to the centroid are normalized based on the maximum distance to the centroid  $M$ . Normalization is to compensate for any change of the size of the object by representing the dimensions of the object in *relative* measurement in stead of *absolute* measurement. If  $D_i$  is the distance from one pixel to the centroid, it will become  $\frac{D_i}{M}$  after the normalization.

6. The signature image is thinned by applying the thinning algorithm [ZS84] to the signature image.
7. M-connectivity is evaluated with each pixel so that a signature image of the edge of the object is obtained in polar coordinates.

The edge signature image of the object in Figure 2.13 (the same object in 2.11 (a)) is illustrated in Figure 2.14. The horizontal axis is from 0 to  $2\pi$  and the vertical axis is the normalized distance from edge pixels to the centroid of the object. Points  $A$ ,  $B$ ,  $C$ , and  $D$  are labeled accordingly in the two figures.

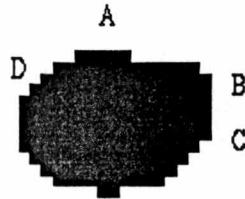


Figure 2.13: Image showing one object with four pixels labeled.

The distance-to-centroid edge signature is invariant to translation, rotation, and scaling. Translation has no effect on the signature since the calculation of the signature does not involve the position of the object in the image. Rotation will cause the curves to shift in the signature image. However, the curves themselves are not changed. Scaling does not affect the signature either since the distances are normalized based on the maximum distance to the centroid.

A signature image is computed for each object in the target dendrone by the above

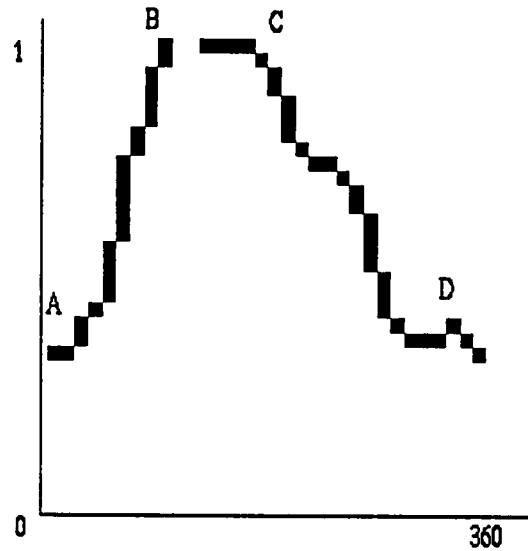


Figure 2.14: The edge signature of the object in Figure 2.13. Four pixels are labeled accordingly.

procedure. The signature images of these target objects are then compared with that of the root object in the source dendrone. The comparison procedure has two steps:

1. If the signature images are not of the same size, the larger image is scaled to be the same size as the smaller one. The scaling is accomplished by re-calculating every pixel's coordinates of the larger signature image. For each pixel  $(x_i, y_i)$  in the larger signature image, a new pixel  $(x'_i, y'_i)$  is calculated as:

$$x'_i = \frac{s_1}{s_2} \times x_i, y'_i = \frac{s_1}{s_2} \times y_i,$$

where

$s_1$  is the dimension of the smaller image, and

$s_2$  is the dimension of the larger image.

2. The signature images are compared and a similarity value between 0 and 1 is calculated to indicate how similar these two objects are. A similarity value of 1 indicates that the two objects are exactly the same and a similarity value of 0 indicates they have no similarities.

Since the signature images are invariant to translation and scaling, a simple *column-by-column* matching scheme is sufficient to compare the two images. In order to compensate the effect of rotation, one signature is shifted once every column. Every shift generates a signature image of the same object except that it is rotated to a certain degree. Every generated signature image for one object is compared with the other object's signature image using the column-by-column matching scheme and the best match is considered the similarity between the two objects. Figure 2.16 is the signature image of the object in Figure 2.15, which is the same object in Figure 2.13. By comparing Figure 2.16 with Figure 2.14, it is clear that these two curves are the same but shifted. Without shifting one of the two signature images, the column-by-column matching scheme would obviously not detect the true similarity (see Figure 2.17).

A similarity value is calculated when two signature images are compared column by column. There are two ways to calculate the similarity value:



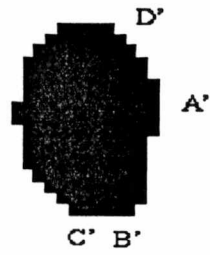


Figure 2.15: One object (rotated object from Figure 2.13) with four pixels labeled.

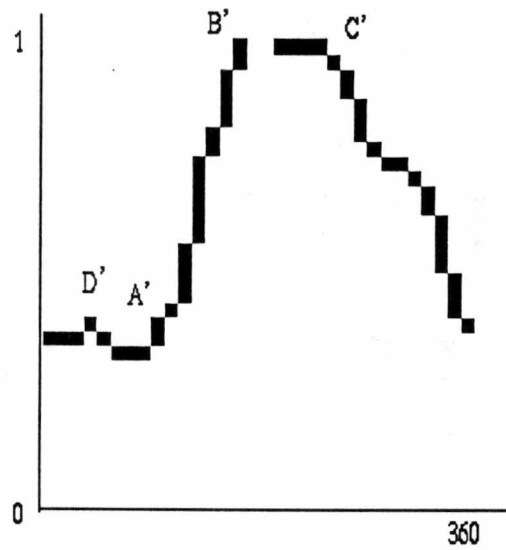


Figure 2.16: The edge signature of the object in Figure 2.15. Four pixels are labeled accordingly.

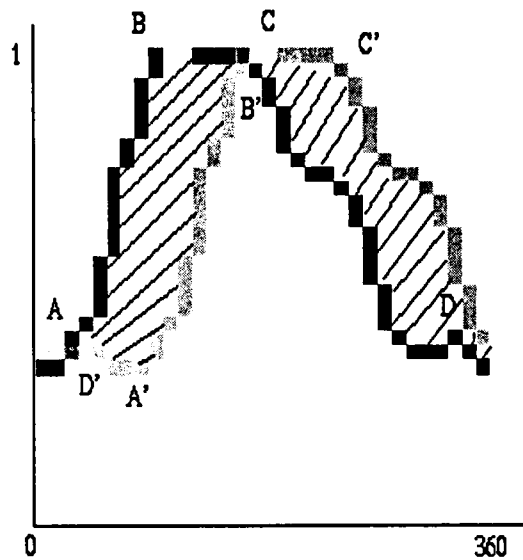


Figure 2.17: Signature image matching.

- (a) The two images are overlapped and the differences between each corresponding pixel of the images are used to compute the similarity value

$$S = 1 - \frac{n_s}{n_t}, \quad (2.2)$$

where

$S$  is the similarity value between the two objects,

$n_s$  is the number of pixels in one image whose corresponding pixels in the other image have the same value, and

$n_t$  is the total number of pixels in one image.

- (b) Each curve in one signature image is compared against each curve in the other

image so that the area between the curves is used to compute the similarity value

$$S = 1 - \frac{\sum_{ij} A_{ij}}{A \times \max(C_1, C_2)}, i = 1, 2, \dots, C_1, j = 1, 2, \dots, C_2, \quad (2.3)$$

where

$S$  is the similarity value between the two objects,

$A_{ij}$  is the area between curve  $i$  in one signature image and curve  $j$  in the other one,

$A$  is the total area of one image,

$C_1$  is the number of curves in one signature image, and

$C_2$  is the number of curves in the other signature image.

For example, in Figure 2.17, the shaded area between the two curves is used to compute the similarity value.

## Chapter 3

# The Design and Implementation of DICE

**DICE** (Dendronic Image Characterization Environment) is an object-oriented computational framework designed for image characterization and retrieval based on dendronic image signatures. The three primary goals of the design and implementation of DICE are *flexibility*, *extendability*, and *portability*. The design of DICE achieves these goals through an object-oriented methodology and using programming languages such as C++ and Java [CW98, Eck98].

Flexibility is achieved so that users of DICE can exploit their own image characterization algorithms. For instance, if the user wants to use a image segmentation algorithm other than the pixel labeling algorithm, he or she can write a method in C++ and override the default algorithm. Alternative techniques for matching objects

from multiple images can also be incorporated into DICE. Extendability is achieved as the user can extend the functionalities of DICE by adding more C++ classes. For example, since the default image format for DICE is XPM [HN91], the user could easily add classes to process images in other formats, such as GIF and JPEG. Portability is achieved because the graphical user interface (or GUI) is implemented in Java which is platform-independent. The GUI can be implemented using other languages or libraries such as X Window/Motif without modifying other modules.

The DICE software environment can be divided into three modules: the dendrone library module (implemented in C++), the GUI module (implemented in Java), and the interface module between the library and the GUI (implemented in C). Figure 3.1 illustrates the interactions among these three modules and the user.

### 3.1 The Dendrone Library

The dendrone library, which is implemented in C++, contains all major functionalities of DICE, such as image segmentation, dendrone construction, and image reconstruction. The dendrone library has eight classes: *Image*, *XpmImage*, *Histogram*, *Option*, *Object*, *Dendrone*, *Shape*, and *RankList*. Figure 3.2 illustrates the class hierarchy and the interactions among the classes.

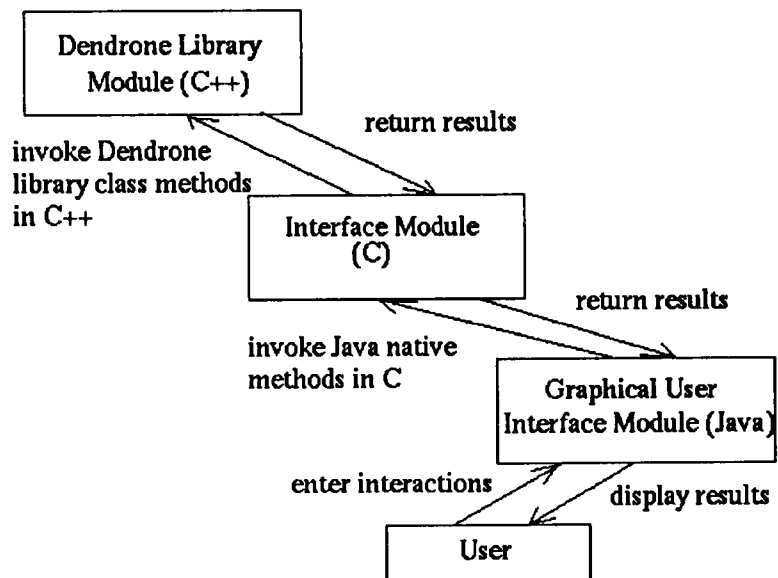


Figure 3.1: Interactions among DICE software modules and the user.

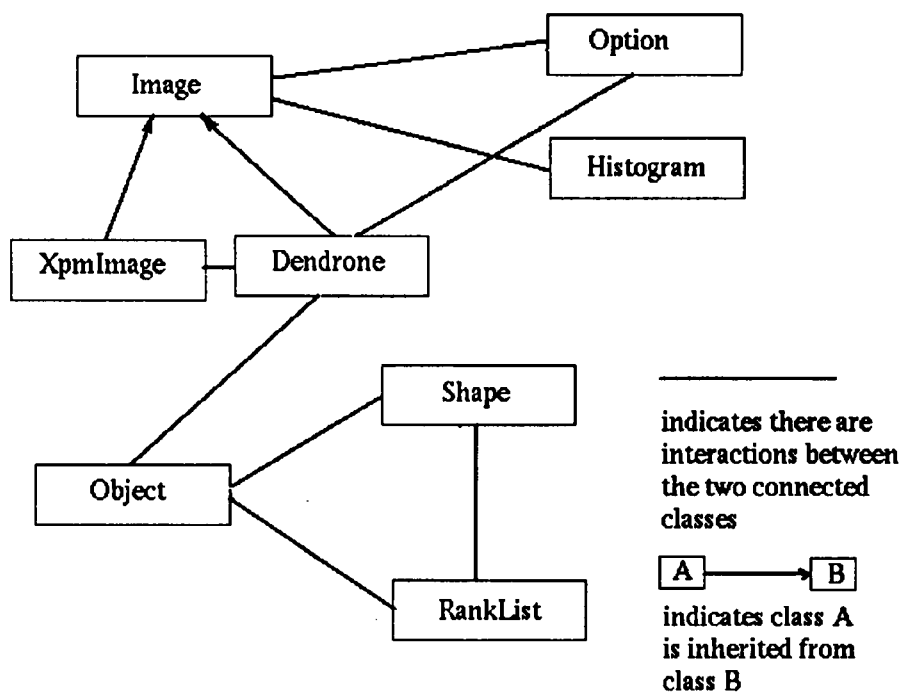


Figure 3.2: Dendrone library class hierarchy.

### **3.1.1 The Image Class**

The *Image* class is the generic class for processing greyscale images. Its functionalities include reading an image into memory, writing an image to a file and so on. Table A.1 in the Appendix illustrates the API (Application Programming Interface) methods of the *Image* class.

### **3.1.2 The XpmImage Class**

The *XpmImage* class, which processes images in XPM format, is a sub-class of the *Image* class. Table A.2 in the Appendix illustrates the API methods of the *XpmImage* class. The user can design new classes for different image formats and make them sub-classes of the *Image* class.

### **3.1.3 The Histogram Class**

The *Histogram* class can calculate and display the histogram, which is the distribution of different intensity values, of an image. Table A.3 in the Appendix illustrates the API methods of the *Histogram* class.

### **3.1.4 The Option Class**

The *Option* class parses command line arguments and sets parameters and control flags for subsequent program execution. The initial design of DICE used a command line interface, which can still be used in shell scripts for batch execution. After the GUI was



incorporated, the *Option* class merely served as a class for passing parameters. Tables A.4 and A.5 in the Appendix illustrate the API methods of the *Option* class.

### 3.1.5 The Object Class

The *Object* class represents the objects in an image. Its functionalities include the ability of checking whether one object touches another object, building the dendrone hierarchy, retrieving pixels from the object, and measuring the similarity of object surface structures. Currently, the *Object* class only stores information about the pixels contained in the object and the dimension and position of the object within the image. If needed, more information such as the ellipse (roundness) of the object could be computed and stored as well. Tables A.6 and A.7 in the Appendix illustrate the API methods of the *Object* class.

### 3.1.6 The Dendrone Class

The *Dendrone* class, which is a sub-class of the *Image* class, represents the dendrones generated from images. Its primary functionalities are segmenting the input image, linking objects generated from the segmentations together to build the dendrone, reconstructing individual object images from the dendrone, and generating PostScript dendrograms. Tables A.8, A.9, and A.10 in the Appendix illustrate the API methods of the *Dendrone* class. Method *buildDendronalTree()* is one of the major methods of the dendrone class, which constructs a dendrone from the input image. Figure 3.3 illustrates the four major steps associated with dendrone construction: image segmentation,

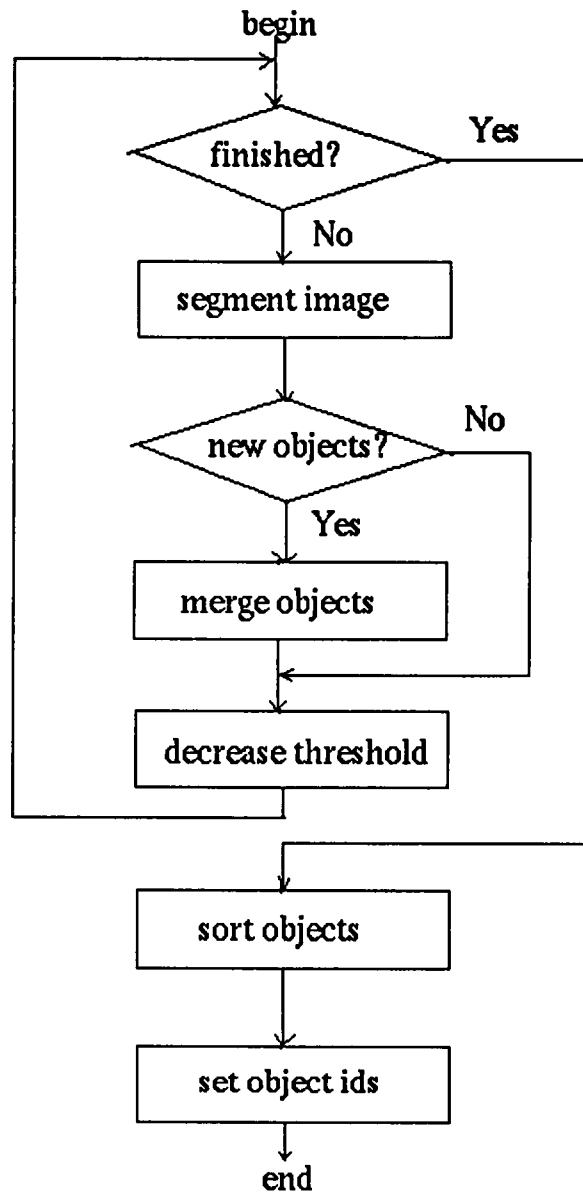


Figure 3.3: Steps of the construction of a dendrone.

object merging, object sorting, and object indexing (declaring object ids).

### **Design of the Dendrone Data Structure**

In the design and implementation of the dendrone data structure, a dynamically linked tree structure is used. Considering the variety of images and the large amount of information a dendrone could contain, static memory allocation is clearly neither practical nor efficient. In addition, during the construction of the dendrone, every iteration of the segmentation process generates a large number of objects. Some of these objects may be merged immediately with other objects while others may not be merged until much later. Furthermore, the merging process requires efficient access to the parent objects and child objects. After the dendrone is built, the objects are sorted according to intensity value. In order to retrieve the objects quickly, and at the same time, without using a lot of memory, a pointer hierarchy is implemented (illustrated in Figure 3.4). The first set of pointers give immediate access to the objects in the dendrone through the ids of the objects. On the other hand, the links among related objects give easy access to parent objects and child objects. A third set of pointers group objects with different levels of intensities together, which are used during the segmentation and merging processes and also facilitate parallelization of the segmentation process.

### **Dendrogram**

For each dendrone, DICE can generate three kinds of dendrograms in PostScript format:

1. The **non-coordinate** dendrogram (see Figure 2.4 for an example).

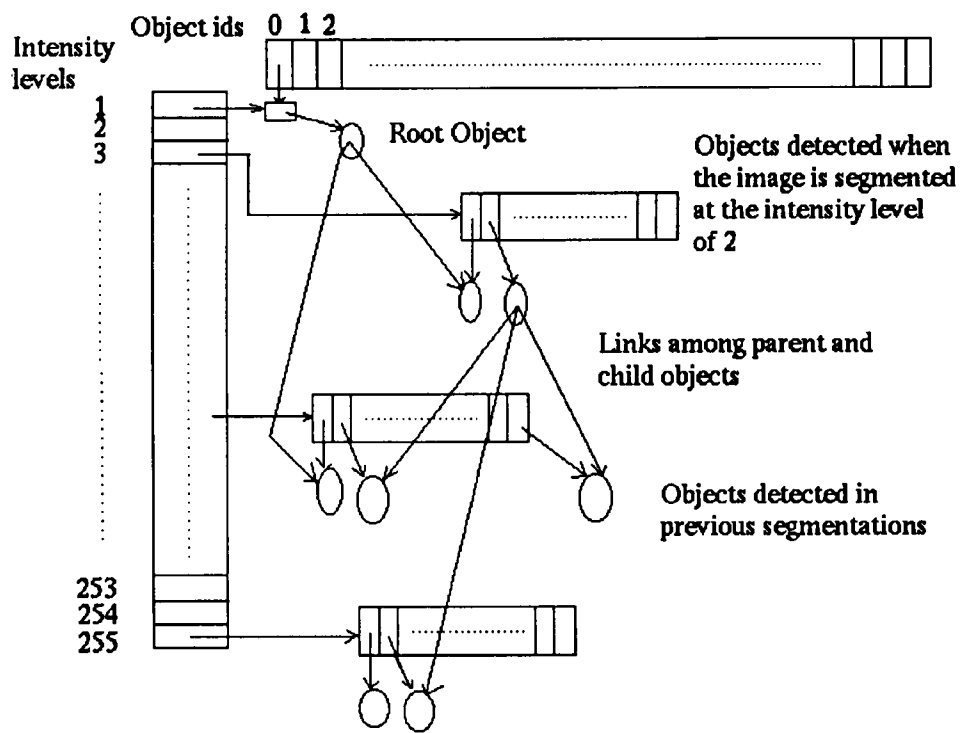


Figure 3.4: Dendrone data structure implementation.

The  $x$ -coordinate of this dendrogram has no meaning and the  $y$ -coordinate indicates the intensity level (ranging from 0 to 255). In addition, the objects in the dendrone are displayed in a certain order:

- (a) Objects with higher intensity values (brighter objects) are displayed to the left side of objects with lower intensity values (darker objects).
- (b) At the same intensity level, composite objects are displayed to the left side of primitive objects.

2. The  **$x$ -coordinate** dendrogram (see Figure 3.5).

The  $x$ -coordinate of this dendrogram corresponds to the  $x$ -coordinate of the image from which the dendrogram is generated. The  $y$ -coordinate indicates the intensity level (ranging from 0 to 255). The individual horizontal and vertical lines in the  $x$ -coordinate dendrogram are interpreted as in the non-coordinate dendrogram.

3. The  **$y$ -coordinate** dendrogram (see Figure 3.6).

The  $y$ -coordinate of this dendrogram corresponds to the  $y$ -coordinate of the image from which the dendrogram is generated, and the  $x$ -coordinate indicates the intensity level. As with the  $x$ -coordinate dendrogram, the horizontal and vertical lines in the  $y$ -coordinate dendrogram designate objects in the image and the intensity levels at which the image is segmented, respectively.

The non-coordinate dendrogram is useful because it does not contain information related to objects' positions within the image, which makes it ideal for comparing den-

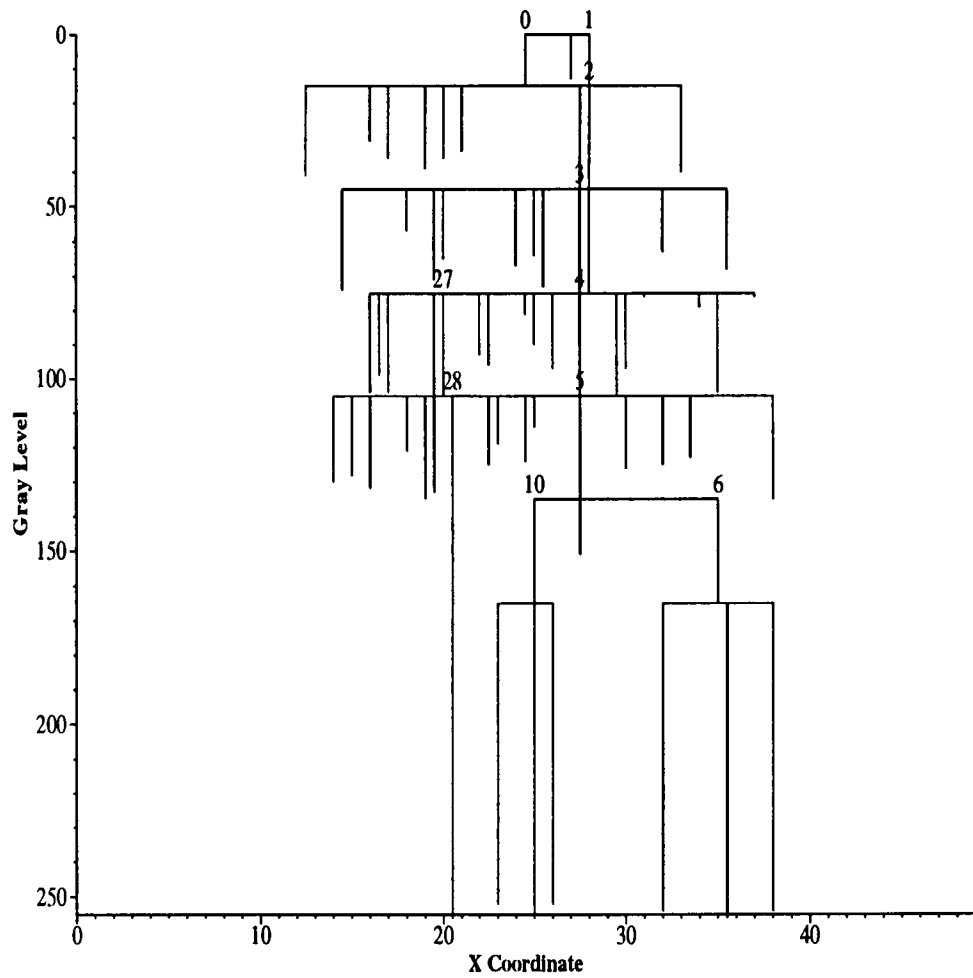


Figure 3.5: X-coordinate dendrogram of the image in Figure 2.1.

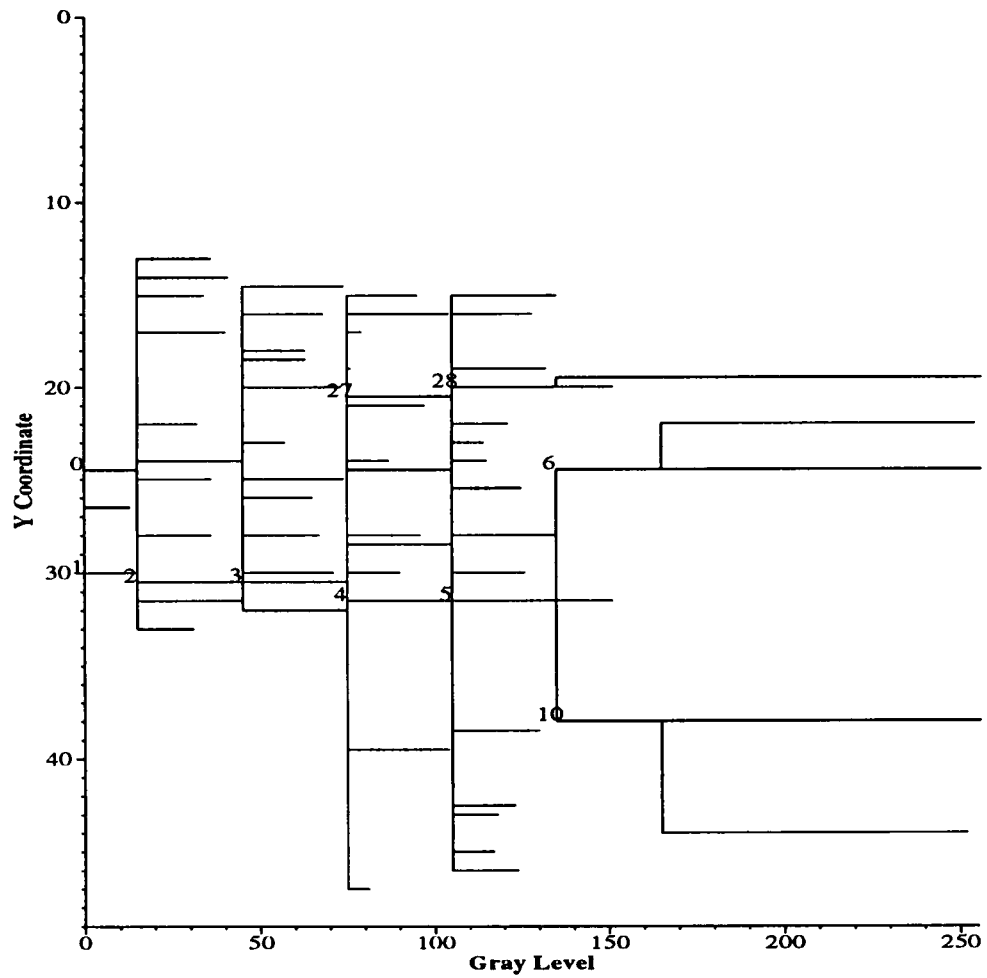


Figure 3.6: Y-coordinate dendrogram of the image in Figure 2.1.

drones generated from rotated or scaled images. The  $x$ -coordinate and  $y$ -coordinate dendrograms are useful when constructing individual object images from the dendrone. From these two kinds of dendrograms, the user can locate an object according to its  $x$ -coordinate and/or  $y$ -coordinate position within the original image (DICE's GUI has a more convenient way to locate objects (see Section 3.2)). As in the non-coordinate dendrogram, composite objects in the  $x$ -coordinate and  $y$ -coordinate dendrograms are identified by the integers near the lines representing the objects.

In Section 2.1, the construction of dendrones was *visualized* as decreasing water level on an imaginary three-dimensional intensity terrain. This approach works well with images that have dark background and bright foreground. If the image has bright background and dark foreground, this approach may not be optimal. In this case, increasing the water level may more appropriately reveal the relationships among the objects in the image. DICE enables both the decrease and increase of the water level so that the user can control how the dendrone should be properly constructed.

### **Text Dendrone File**

In DICE, the user can save the dendrones generated from images to files for future processing. A specific text format has been designed to store dendrones. This format is easy to modify for test purposes and can be compressed using external programs.

In the file, comment lines begin with a “#” and are ignored by DICE. The first meaningful line/record must be of the form



*maxID rows cols left top right bottom stride reverse* ,

where

*maxID* is the maximum object id in the dendrone,

*rows* is the number of rows in the image,

*cols* is the number of columns in the image,

*left*, *top*, *right*, and *bottom* are the coordinates of the sub-image from which  
the dendrone is generated,

*stride* is the stride value at which the image is segmented, and

*reverse* indicates whether the dendrone is generated from the highest  
intensity value to the lowest or vice versa.

The remaining lines/records of the file contain information of the objects in the  
dendrone. The objects are listed according to a pre-order traversal of the dendrone.

Each composite object is listed as

$(ID, top, left, bottom, right, cRow, cCol, size, subObjNo) \rightarrow$  ,

where

*ID* is the id of the object,

*top*, *left*, *bottom*, and *right* are the coordinates of the object's bounding box,  
*cRow* is the *y*-coordinate of the center of the bounding box,  
*cCol* is the *x*-coordinate of the center of the bounding box,  
*size* is the number of pixels in the object, and  
*subObjNo* is the number of child objects.

Primitive objects are listed under their parent objects respectively, each followed by information about the pixels in the object:

$$\begin{aligned}
 &(ID, top, left, bottom, right, cRow, cCol, size, 0) \\
 &\quad (y_1, x_1, intensity_1) \\
 &\quad (y_2, x_2, intensity_2) \\
 &\quad \dots \\
 &\quad (y_{size}, x_{size}, intensity_{size})
 \end{aligned}$$

where

$y_i$  ( $i \in [1, size]$ ) is the *y*-coordinate of one pixel in the object,  
 $x_i$  ( $i \in [1, size]$ ) is the *x*-coordinate of one pixel in the object, and  
 $intensity_i$  ( $i \in [1, size]$ ) is the intensity value of the pixel.

## Reconstructed Image

In DICE, the user can reconstruct individual object images from the dendrone generated from the input image. The generated images are in PPM [Pos91] format. Currently, the reconstructed objects are rendered using their ids as the intensity values. In the

dendrone library, it is also possible to render the objects using their original intensity values.

## **File Management**

For one input image, DICE may generate a large number of output files in different formats. To manage these files, a simple *root filename* approach is taken. A root filename is chosen (either by default or designated by the user) for each input image file and certain attributes and extension names are appended to the root filename to form the output filename. Table 3.1 summarizes the file management in DICE. The Jgraph files shown are used to generate PostScript dendrograms. Jgraph is a simple description language for plotting graphs (see [Pla]).

### **3.1.7 The Shape Class**

The *Shape* class implements the distance-to-centroid image signature algorithms presented in Section 2.2. It can generate the signature image from a detected object and compare similarity of object shapes. This class can be replaced if the user chooses an alternative object matching algorithm. Table A.11 in the Appendix illustrates the API methods of the *Shape* class.

### **3.1.8 The RankList Class**

The *RankList* class collects the results of object matching into a ranked list sorted by similarity value in descending order and handles information passing between the GUI

Table 3.1: File management in DICE.

File Name	Format	Description	Example
root.xpm	XPM	input image file	peaks.xpm
root- <i>stride-objID</i> .txt	ASCII text	text (sub-)dendrone file	peaks-30-0.txt
root-histo.ps	PostScript	PostScript histogram	peaks-histo.ps
root- <i>stride-objID</i> .jgr	Jgraph	Jgraph non-coordinate (sub-)dendrogram	peaks-30-0.jgr
root-x- <i>stride-objID</i> .jgr	Jgraph	Jgraph <i>x</i> -coordinate (sub-)dendrogram	peaks-x-30-0.jgr
root-y- <i>stride-objID</i> .jgr	Jgraph	Jgraph <i>y</i> -coordinate (sub-)dendrogram	peaks-y-30-0.jgr
root- <i>stride-objID</i> .ps	PostScript	PostScript non-coordinate (sub-)dendrogram	peaks-30-0.ps
root-x- <i>stride-objID</i> .ps	PostScript	PostScript <i>x</i> -coordinate (sub-)dendrogram	peaks-x-30-0.ps
root-y- <i>stride-objID</i> .ps	PostScript	PostScript <i>y</i> -coordinate (sub-)dendrogram	peaks-y-30-0.ps
root- <i>stride-objID</i> .ppm	PPM	reconstructed object image	peaks-30-0.ppm

and the dendrone library. Table A.12 illustrates the API methods of the *RankList* class.

## 3.2 The Graphical User Interface

The GUI module (see Figure 3.7), implemented in Java, provides convenient ways for the user to generate dendrones from images, reconstruct images from dendrones, and retrieve objects in images by matching sub-dendrones. The GUI also provides a customizable configuration environment, which the user can modify according to personal preferences.

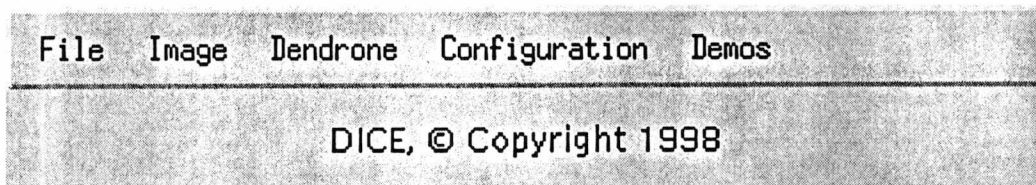


Figure 3.7: *DICE* main window.

### 3.2.1 Generating Dendrones from Images

In the GUI, it is straightforward to generate dendrones from images. First, the user selects an input image from a *FileDialog*. If the file is successfully read, a *ImageDisplayer* window (see Figure 3.8) containing the image is displayed. In the *ImageDisplayer* window, the user can select a sub-image (represented by the rectangle) by clicking and dragging the mouse button. Without a selected sub-image, the dendrone will be built from the entire image. By clicking the *Build Dendrone...* button, the user can bring up a *BuildDialog* window (see Figure 3.9) in which a number of parameters and options regarding the construction of the dendrone can be selected. The *Output File Root* is automatically derived from the input image file name unless supplied by the user. If the root file name is not an absolute path name, the generated files (see Table 3.1) will be stored in the directory relative to the current directory. The user can also set the stride value using the scroll bar and edit the *Left*, *Top*, *Right*, and *Bottom* text fields for the sub-image coordinates by either entering numbers manually or selecting a sub-image in the *ImageDisplayer* window using the mouse. The radio buttons and check boxes allow the user to generate and display selected PostScript dendrograms.

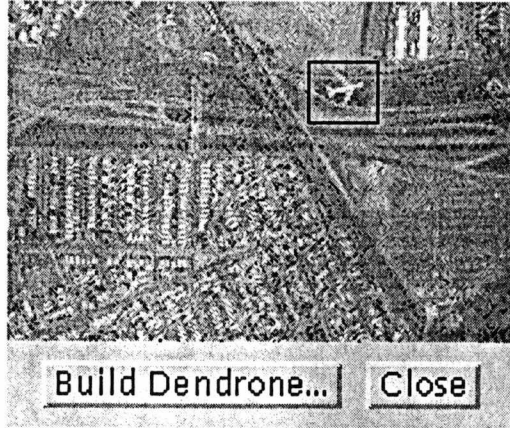


Figure 3.8: *ImageDisplayer* window for dendrone construction.

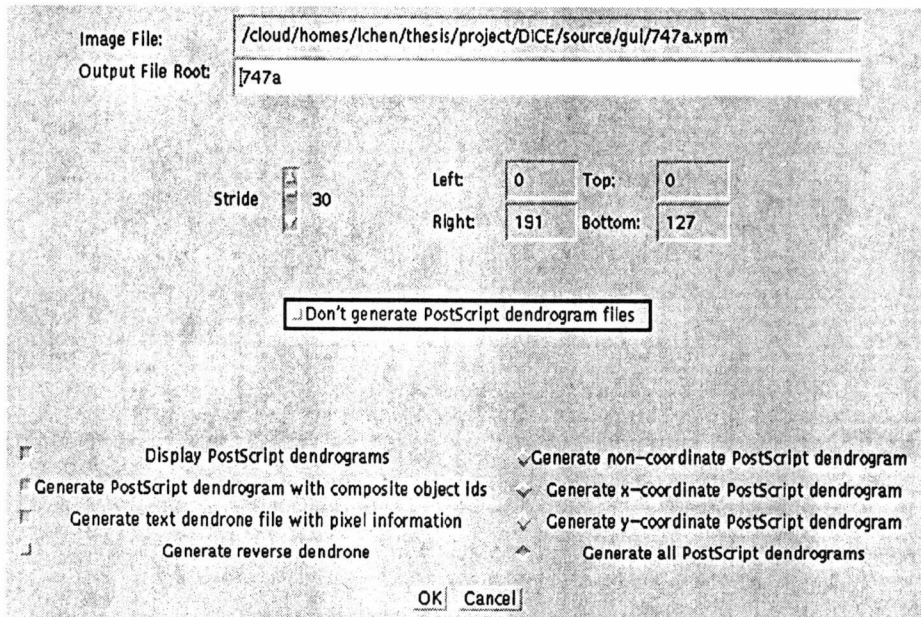


Figure 3.9: *BuildDialog* window for dendrone construction.

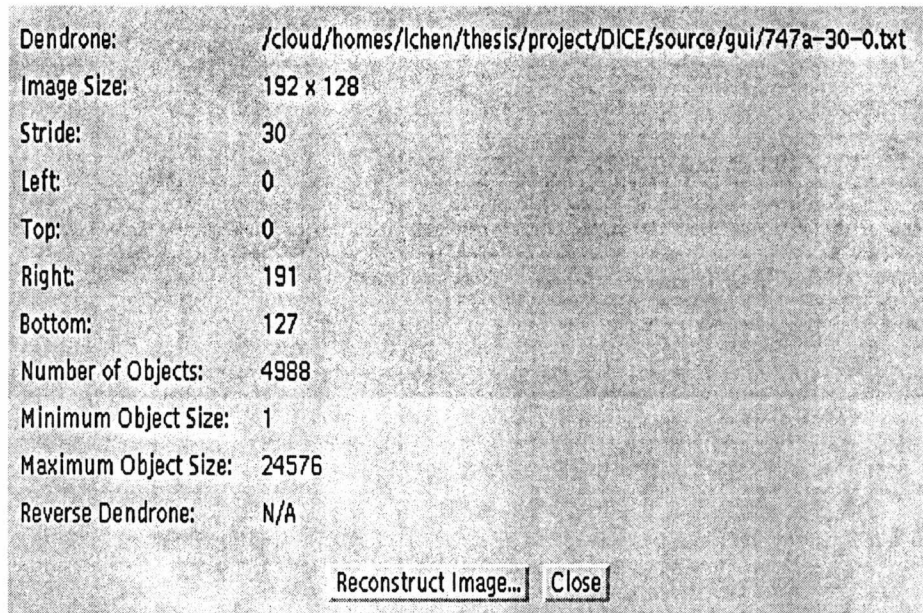


Figure 3.10: *DendroneDisplayer* window.

### 3.2.2 Reconstructing Images from Dendrones

With the help of the GUI, reconstructing images from dendrones becomes very convenient. The user first loads the dendrone into DICE by selecting a file storing the dendrone using the *FileDialog*. If the dendrone is successfully loaded, a *DendroneDisplayer* window (see Figure 3.10) will be displayed. This window contains information about the dendrone, such as the size and coordinates of the (sub-)image from which the dendrone is generated and the stride value used to generate the dendrone. Clicking the *Reconstruct Image...* button will bring up a *ReconstructDialog* window (see Figure 3.11) for specifying additional information concerning the reconstruction. An *ImageDisplayer* window (see Figure 3.12) containing the original (sub-)image is also displayed.

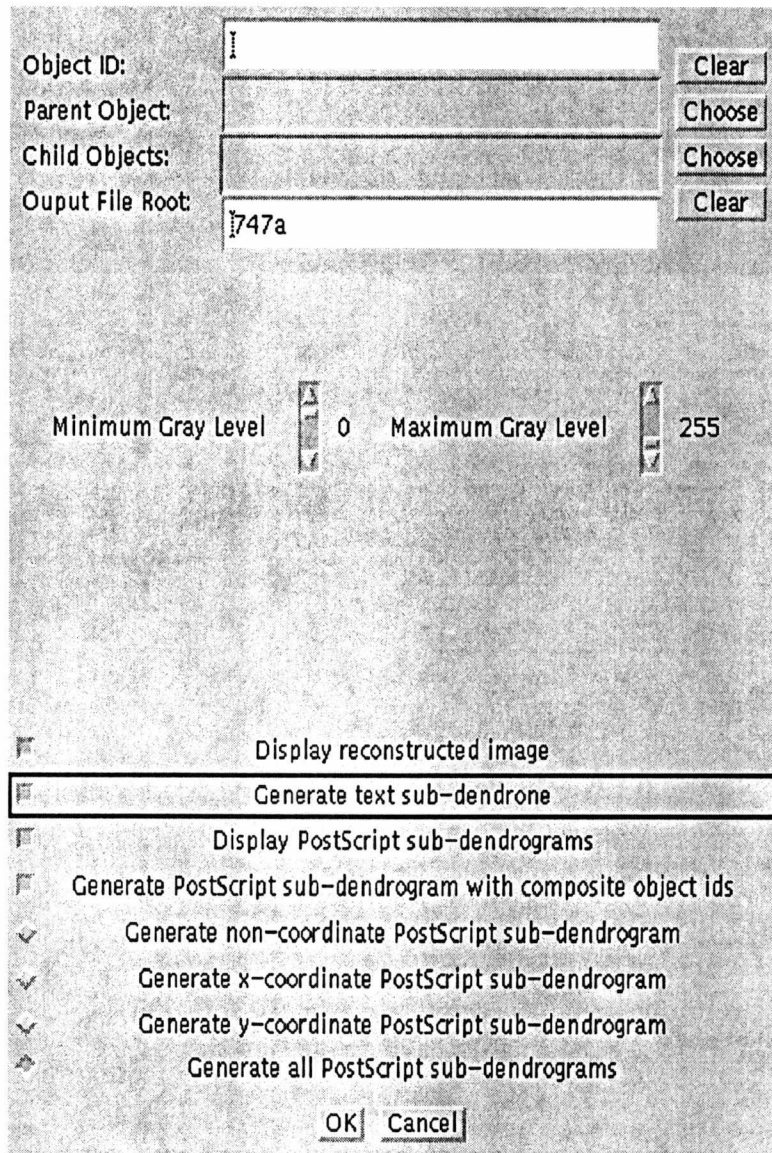


Figure 3.11: *ReconstructDialog* window for image reconstruction.



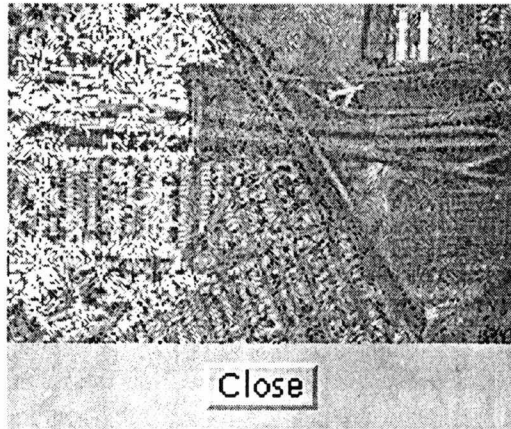


Figure 3.12: *ImageDisplayer* window for image reconstruction.

This window is identical to the one used to generate dendrones from images (illustrated in Figure 3.8) with the exception that the *Build Dendrone...* button is not present.

The reconstruction of an individual object is accomplished by using its id. A few different ways to obtain an object's id are:

1. Point and click the mouse in the *ImageDisplayer* window.

This is the most convenient way to locate an object. The user can move the mouse to a pixel and click on the pixel. The *smallest* object containing that pixel will be highlighted (see Figure 3.12 for an example). At the same time, the *Object ID* text field, the *Parent Object* text field, and the *Child Objects* scroll list are updated accordingly.

2. Enter the object id in the *Object ID* text field directly.

If the user knows the object's id (by viewing the PostScript dendrograms, for

example), he or she can enter the id directly. The corresponding object with that id in the *ImageDisplayer* window will be highlighted. In addition, the *Parent Object* text field and the *Child Objects* scroll list are updated automatically to contain the parent object id and child object ids, respectively.

3. Select an object id from the *Parent Object* text field or the *Child Objects* scroll list.

After the initial object is selected (for instance, by pointing and clicking the mouse), the user can traverse the dendrone by using the *Parent Object* text field and the *Child Objects* scroll list until the desired object is found.

In the *ReconstructDialog* window, the user can also specify the root file name for output files, a range of intensity values for reconstruction, and other options regarding the generation and display of object images, text sub-dendrone files, and PostScript sub-dendrograms.

### 3.2.3 Object Retrieval by Matching Sub-Dendrones

The *MatchDialog* window (see Figure 3.13) can be used to retrieve objects with similar surface structure and/or shape as determined by matching sub-dendrones representing the objects. The user can enter the name of the file containing the dendrone representing the source object to match and the name of the file containing the target image from which objects will be retrieved. To narrow down the number of potential candidate objects, the user can increase the similarity value, or specify the size of the objects to

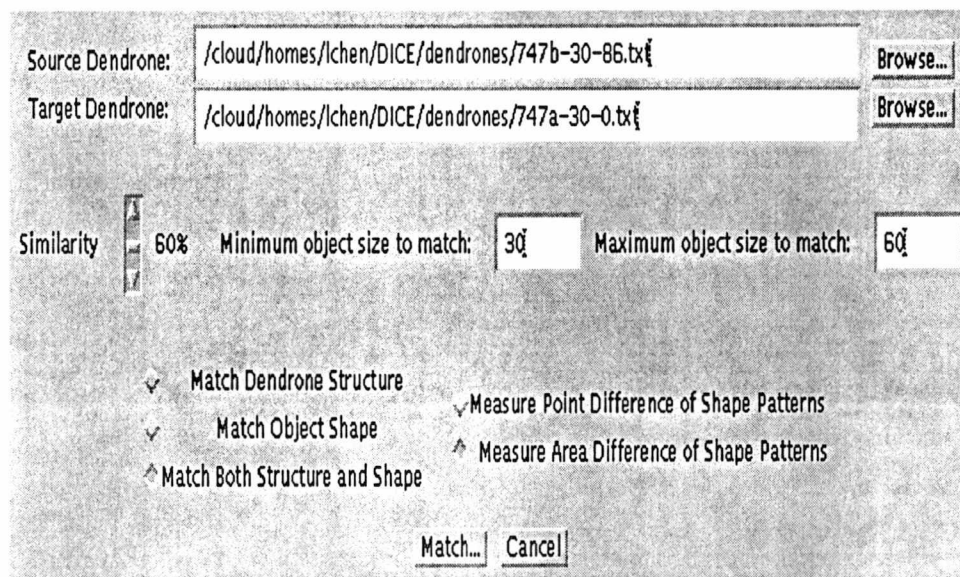


Figure 3.13: *MatchDialog* window for object matching.

retrieve. Finally, the user can select the criteria for the match. Currently, DICE only implements two kinds of matching - matching by dendrone structure and by object shape using the distance-to-centroid signature. The user can write his or her own *MatchDialog* class in Java to implement alternative matching algorithms. After the user clicks the *Match...* button, the matching results will be displayed in a *MatchResultDisplayer* window (see Figure 3.14). The results are sorted by the similarity value in descending order. The ids and sizes of the matching objects are also displayed so that the user can double-click on an item to bring up an *ReconstructDialog* window and reconstruct the selected object's image.

Rank	Object ID	Similarity	Size
1	257	71.81%	40
2	458	71.52%	43
3	930	67.33%	37
4	958	65.12%	30
5	1794	62.38%	53

Reconstruct Image... Close

Figure 3.14: *MatchResultDisplayer* window showing object matching results.

### 3.2.4 Customizing DICE

DICE provides a configuration file which allows each individual user to set his or her own default DICE environment. The user can either edit the text configuration file directly or use the DICE GUI. As illustrated in Figure 3.15, the user can specify any available application programs to display and edit images. The parameters and options used in the construction of dendrones, reconstruction of images, and matching of sub-dendrones are also customizable so that the user doesn't have to set the same options redundantly. To demonstrate the use of the distance-to-centroid matching algorithm, DICE has included a sub-dendrone matching demo, which matches 747 airplanes in two different images.

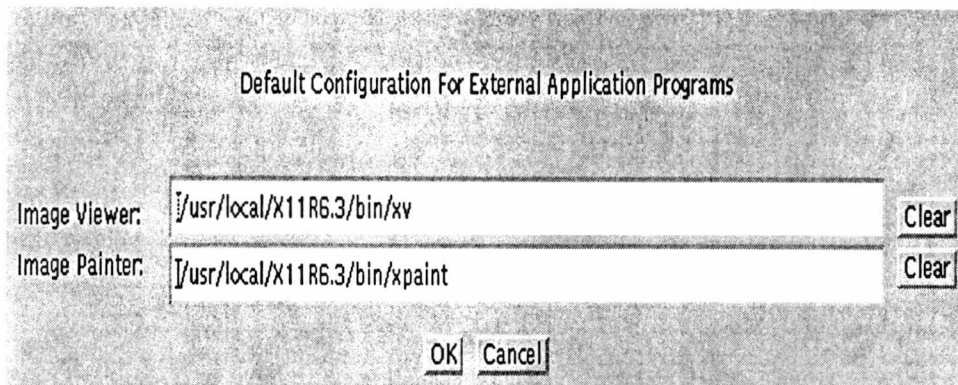


Figure 3.15: *AppConfigDialog* window for external application program configuration.

### 3.3 The Interface Module

Since the dendrone library and the GUI module are written in two different programming languages, an interface module must be used to communicate between them. More specifically, in the GUI module, Java objects need to call methods in the C++ dendrone library. DICE uses the *JNI* (Java Native Interface) mechanism [CW98, Eck98] provided by Sun's JDK (Java Development Kit) to accomplish the interactions between Java objects and C++ methods. The interface module, written in C, contains *wrapper* functions which call a C++ method and return results to Java objects. Such functions are required since the JDK only supports native methods written in C. In order to call a C++ method, a C function must be used, which accepts a pointer to a C++ object for subsequent method invocations. Figure 3.16 illustrates the steps of calling a C++ method from a Java object:

1. The Java object calls the C function, passing a pointer to the designated C++

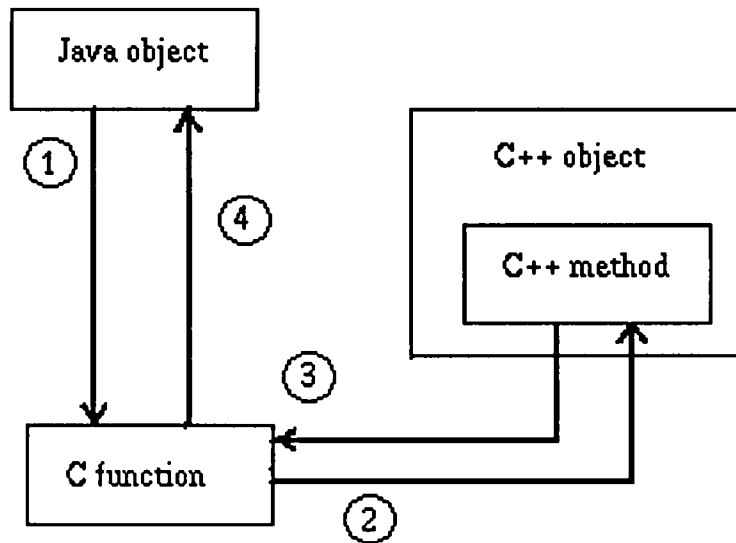


Figure 3.16: Calling C++ methods in Java.

object, which is returned from a previous invocation of another C++ method, and other parameters.

2. The C function calls the C++ method in the designated object.
3. The C++ method returns results to the C function.
4. The C function returns results to the Java object.

## Chapter 4

# Evaluating the Performance and Effectiveness of DICE

The focus of this chapter is to evaluate the performance and effectiveness of the DICE software. The benchmarked execution time and complexity of constructing dendrones from images are discussed. The effectiveness of the object matching algorithm using distance-to-centroid image signatures is also examined.

### 4.1 Performance Evaluation

To evaluate the performance of dendrone construction algorithms, processing times for a few benchmark images (of different sizes and contents) were recorded.

### 4.1.1 Procedure

All performance timings were recorded on a Sun Ultra1 SPARCstation with a 167 MHz processor, 32 KB on-chip cache (16 KB Instruction, 16 KB Data), 512 KB external cache, and 256 MB of main memory. A C version of the DICE dendrone library software (presented in Section 3.1) exploiting a command line interface was used for the evaluation. As illustrated in Table 4.1 and Figures 4.1 and 4.2, two different images were used in the experiment.

Table 4.1: Images used to evaluate the performance of the dendrone construction algorithms.

Figure	Size in Pixels (rows $\times$ columns)	Description
4.1	128 $\times$ 192	747 airplane
4.2	264 $\times$ 472	biplane



Figure 4.1: High-altitude photograph of a Boeing 747 in flight.

As mentioned in Section 2.2.1, three different image segmentation algorithms are available:



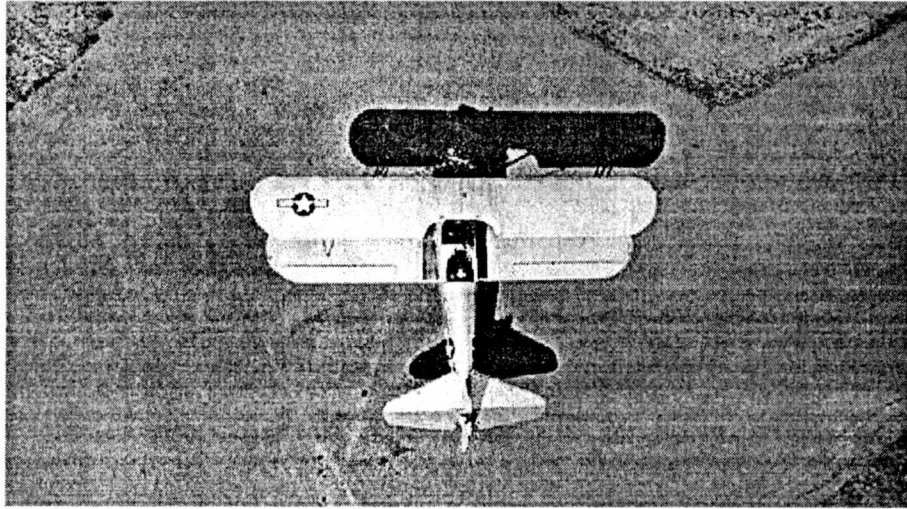


Figure 4.2: A biplane and its shadow as seen from above.

1. The pixel labeling algorithm.
2. The recursive version of Connectivity Filling.
3. The non-recursive version of Connectivity Filling.

One implementation of each of the three image segmentation algorithms was benchmarked. In the pixel labeling version, elapsed CPU times were recorded in four key steps of the *buildDendronalTree()* function (see Figure 3.3): image segmentation, object merging, object sorting, and object indexing (i.e., declaring object ids). In the Connectivity Filling version, an extra step, pixel sorting, must be performed before the above four steps. In this step, the pixels in the image are sorted by their intensity values in descending order. To reveal the relationship between the time spent in each of these steps and the stride value used to segment the images and construct the dendrones,

Table 4.2: Summary of dendrone construction performance figures and tables.

Figure/Table	Input Image	Segmentation Algorithm
Figure 4.3	Figure 4.1 (747)	Pixel Labeling
Table 4.3	Figure 4.1 (747)	Pixel Labeling
Table 4.4	Figure 4.1 (747)	Pixel Labeling
Figure 4.4	Figure 4.1 (747)	Recursive Connectivity Filling
Table 4.5	Figure 4.1 (747)	Recursive Connectivity Filling
Table 4.6	Figure 4.1 (747)	Recursive Connectivity Filling
Figure 4.5	Figure 4.1 (747)	Non-recursive Connectivity Filling
Table 4.7	Figure 4.1 (747)	Non-recursive Connectivity Filling
Table 4.8	Figure 4.1 (747)	Non-recursive Connectivity Filling
Figure 4.6	Figure 4.2 (Biplane)	Pixel Labeling
Table 4.9	Figure 4.2 (Biplane)	Pixel Labeling
Table 4.10	Figure 4.2 (Biplane)	Pixel Labeling
Figure 4.7	Figure 4.2 (Biplane)	Non-recursive Connectivity Filling
Table 4.11	Figure 4.2 (Biplane)	Non-recursive Connectivity Filling
Table 4.12	Figure 4.2 (Biplane)	Non-recursive Connectivity Filling

different stride values ranging from 1 up to 255 were used.

#### 4.1.2 Results

The elapsed times were collected and drawn in Figures 4.3 through 4.7. To illustrate the percentage of time consumed by different steps in the dendrone construction, the total construction time is broken down at selected stride values. Table 4.2 summarizes these figures and tables.

From Figures 4.3 through 4.7, it is clear that the smaller the stride value, the longer the total time of dendrone construction. As the stride value is chosen to be smaller, relatively more objects are detected during the segmentation processes, although the

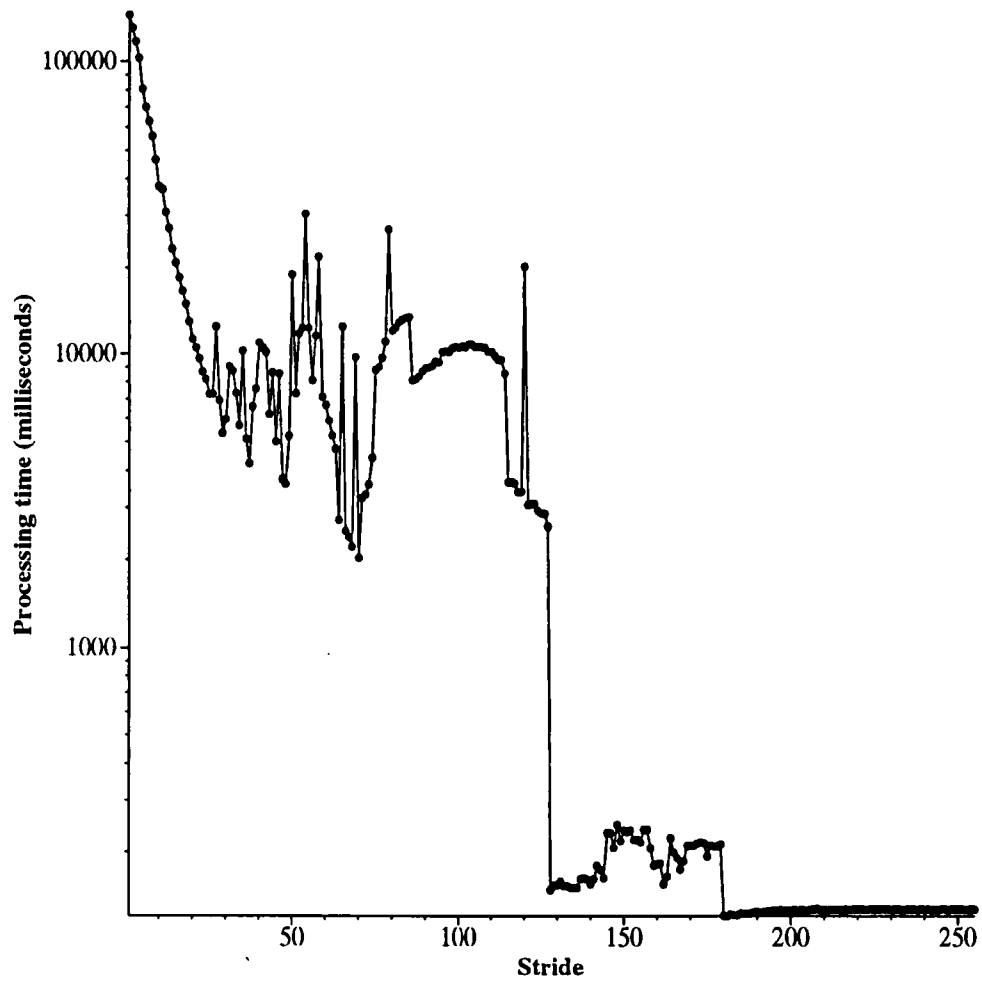


Figure 4.3: Total elapsed time of dendrone construction for the image in Figure 4.1 using the pixel labeling algorithm.

Table 4.3: Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.1 using the pixel labeling algorithm.

Stride	Segmentation	Merging	Indexing	Object Sorting	Total
1	3149.87	141301.97	64.00	187.70	144715.61
5	850.46	77834.05	52.04	1826.38	80565.80
10	562.91	36098.59	43.54	1077.91	37784.54
20	412.66	10128.57	20.39	732.91	11295.34
30	351.38	5044.24	12.04	573.77	5982.01
50	269.82	18425.86	5.72	258.25	18960.06
100	161.03	4192.02	2.20	6169.51	10525.00
150	113.98	0.00	0.26	122.68	237.13
200	127.38	0.00	0.04	0.07	127.71
255	128.04	0.00	0.04	0.07	128.38

Table 4.4: Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.1 using the pixel labeling algorithm.

Stride	Segmentation	Merging	Indexing	Object Sorting
1	2.18	97.64	0.04	0.13
5	1.06	96.61	0.06	2.27
10	1.49	95.54	0.12	2.85
20	3.65	89.67	0.18	6.49
30	5.87	84.32	0.20	9.59
50	1.42	97.18	0.03	1.36
100	1.53	39.83	0.02	58.62
150	48.06	0.00	0.11	51.74
200	99.74	0.00	0.03	0.05
255	99.74	0.00	0.03	0.06

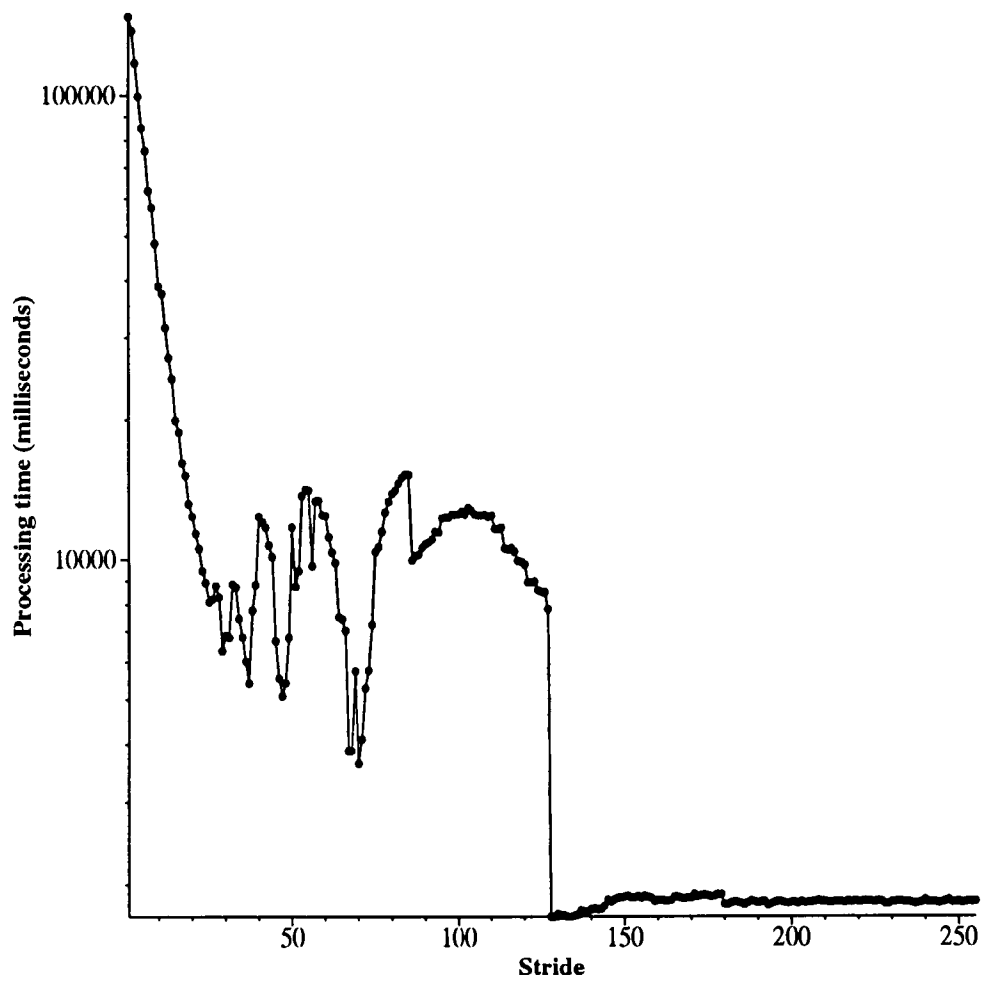


Figure 4.4: Total elapsed time of dendrone construction for the image in Figure 4.1 using the recursive Connectivity Filling algorithm.

Table 4.5: Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.1 using the recursive Connectivity Filling algorithm.

Stride	Pixel Sorting	Segmentation	Merging	Indexing	Object Sorting	Total
1	1604.33	400.18	145563.12	64.56	192.73	147836.19
5	1583.88	366.38	82705.34	51.87	409.38	85120.14
10	1591.96	333.48	36524.52	36.68	301.38	38789.83
20	1588.71	289.42	10171.17	19.39	290.50	12360.31
30	1597.70	268.41	4722.05	12.08	258.63	6859.79
50	1587.09	252.92	6236.29	5.54	3623.13	11705.68
100	1588.05	252.87	4267.07	1.99	6385.33	12495.85
150	1590.80	173.58	0.00	0.22	124.03	1889.12
200	1590.40	246.07	0.00	0.04	0.03	1837.02
255	1597.94	246.69	0.00	0.03	0.03	1845.18

Table 4.6: Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.1 using the recursive Connectivity Filling algorithm.

Stride	Pixel Sorting	Segmentation	Merging	Indexing	Object Sorting
1	1.09	0.27	98.46	0.04	0.13
5	1.86	0.43	97.16	0.06	0.48
10	4.10	0.86	94.16	0.09	0.78
20	12.85	2.34	82.29	0.16	2.35
30	23.29	3.91	68.84	0.18	3.77
50	13.56	2.16	53.28	0.05	30.95
100	12.71	2.02	34.15	0.02	51.10
150	84.21	9.19	0.00	0.01	6.57
200	86.57	13.40	0.00	0.00	0.00
255	86.60	13.37	0.00	0.00	0.00

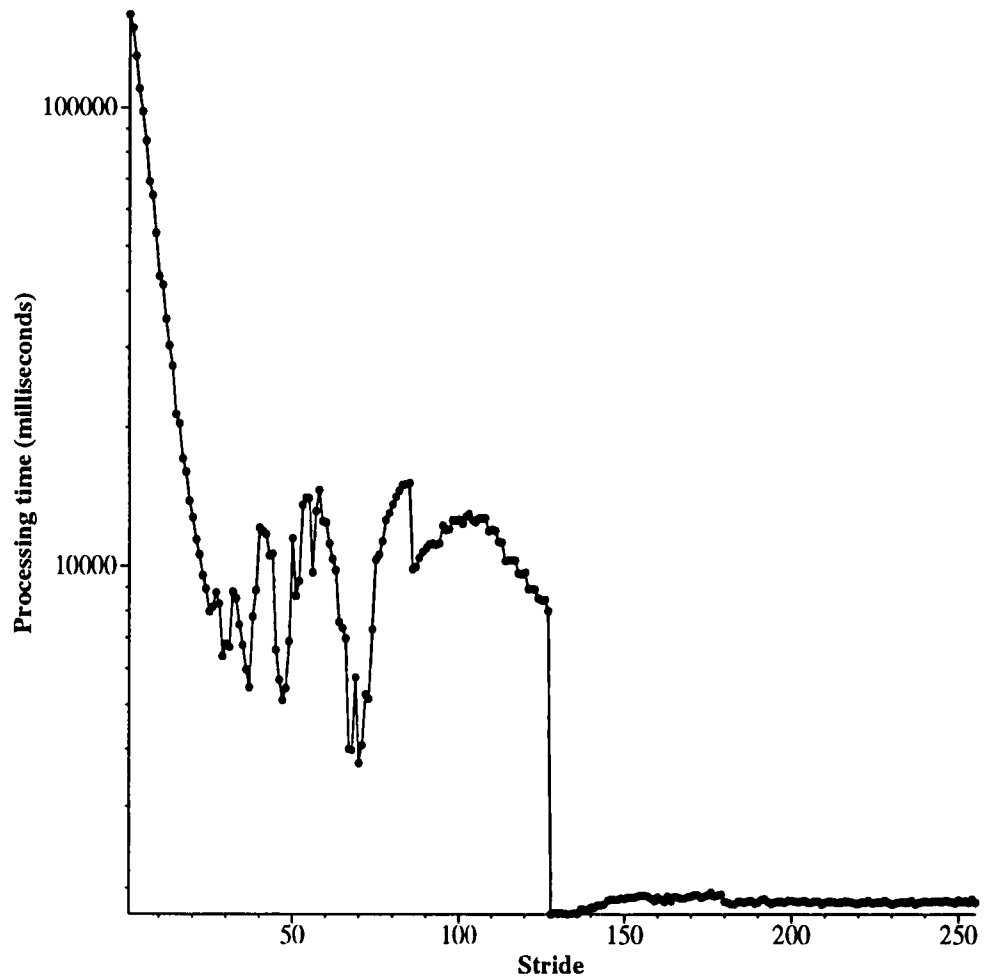


Figure 4.5: Total elapsed time of dendrone construction for the image in Figure 4.1 using the non-recursive Connectivity Filling algorithm.

Table 4.7: Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.1 using the non-recursive Connectivity Filling algorithm.

Stride	Pixel Sorting	Segmentation	Merging	Indexing	Object Sorting	Total
1	1640.32	649.69	157450.41	66.33	198.08	160016.16
5	1601.99	602.69	95345.07	55.46	419.15	98027.55
10	1614.97	513.40	40549.01	37.26	310.46	43026.94
20	1595.05	409.60	10395.39	19.68	296.30	12717.13
30	1613.75	348.80	4574.30	12.10	256.37	6806.23
50	1621.52	292.73	6061.30	5.40	3495.64	11477.28
100	1618.94	264.64	4171.13	2.09	6542.94	12600.32
150	1604.27	164.80	0.00	0.26	127.68	1897.51
200	1628.02	248.95	0.00	0.03	0.03	1877.55
255	1614.94	246.08	0.00	0.03	0.03	1861.57

Table 4.8: Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.1 using the non-recursive Connectivity Filling algorithm.

Stride	Pixel Sorting	Segmentation	Merging	Indexing	Object Sorting
1	1.03	0.41	98.40	0.04	0.12
5	1.63	0.61	97.26	0.06	0.43
10	3.75	1.19	94.24	0.09	0.72
20	12.54	3.22	81.74	0.15	2.33
30	23.71	5.12	67.21	0.18	3.77
50	14.13	2.55	52.81	0.05	30.46
100	12.85	2.10	33.10	0.02	51.93
150	84.55	8.68	0.00	0.01	6.73
200	86.71	13.26	0.00	0.00	0.00
255	86.75	13.22	0.00	0.00	0.00



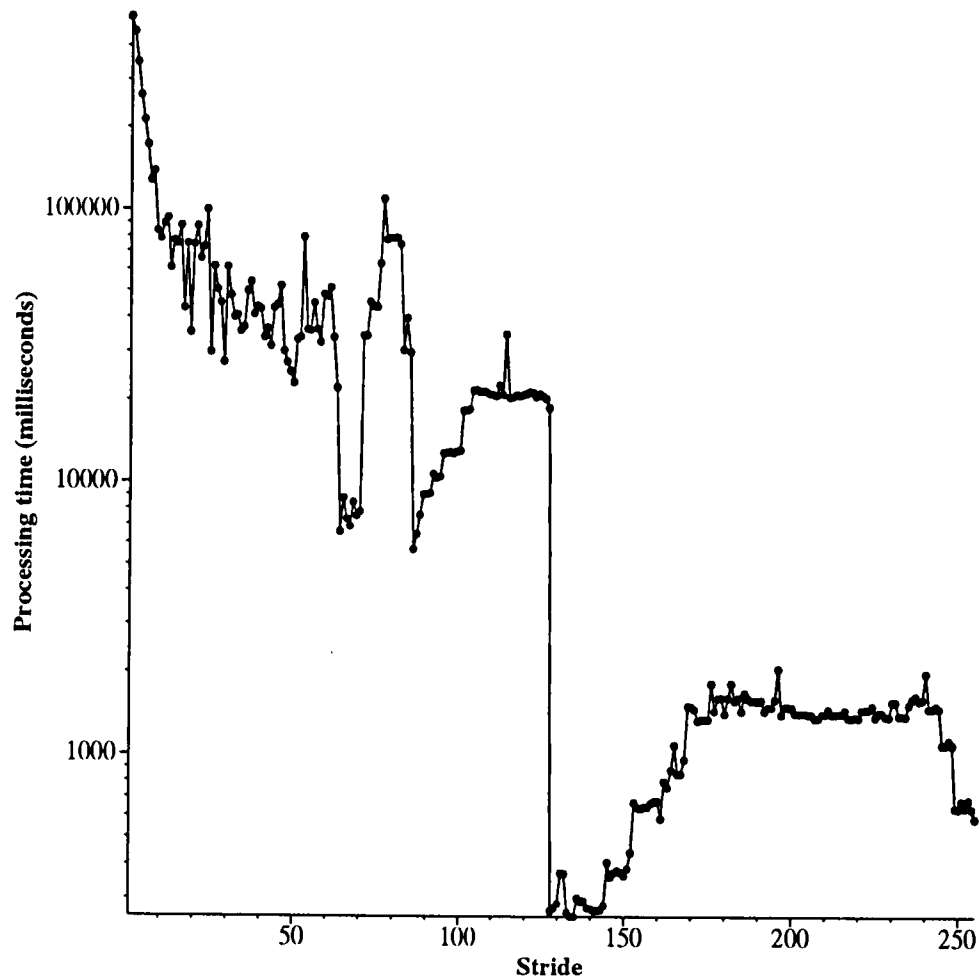


Figure 4.6: Total elapsed time of dendrone construction for the image in Figure 4.2 using the pixel labeling algorithm.

Table 4.9: Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.2 using the pixel labeling algorithm.

Stride	Segmentation	Merging	Indexing	Object Sorting	Total
1	15983.31	491851.16	114.31	1654.03	509615.09
5	4612.73	201469.41	76.69	7113.61	213275.58
10	2781.57	65908.59	39.02	9484.02	78214.82
20	1733.67	70949.89	20.09	1756.38	74460.95
30	1419.86	57862.00	12.78	1881.23	61176.66
50	1046.15	19135.43	4.55	2889.49	23076.08
100	695.25	4046.65	0.65	8322.47	13065.35
150	233.76	0.00	1.40	128.96	364.48
200	610.61	0.00	0.10	875.80	1486.73
255	585.97	0.00	0.03	0.07	586.30

Table 4.10: Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.2 using the pixel labeling algorithm.

Stride	Segmentation	Merging	Indexing	Object Sorting
1	3.14	96.51	0.02	0.32
5	2.16	94.46	0.04	3.34
10	3.56	84.27	0.05	12.13
20	2.33	95.28	0.03	2.36
30	2.32	94.58	0.02	3.08
50	4.53	82.92	0.02	12.52
100	5.32	30.97	0.01	63.70
150	64.13	0.00	0.39	35.38
200	41.07	0.00	0.01	58.91
255	99.94	0.00	0.01	0.01

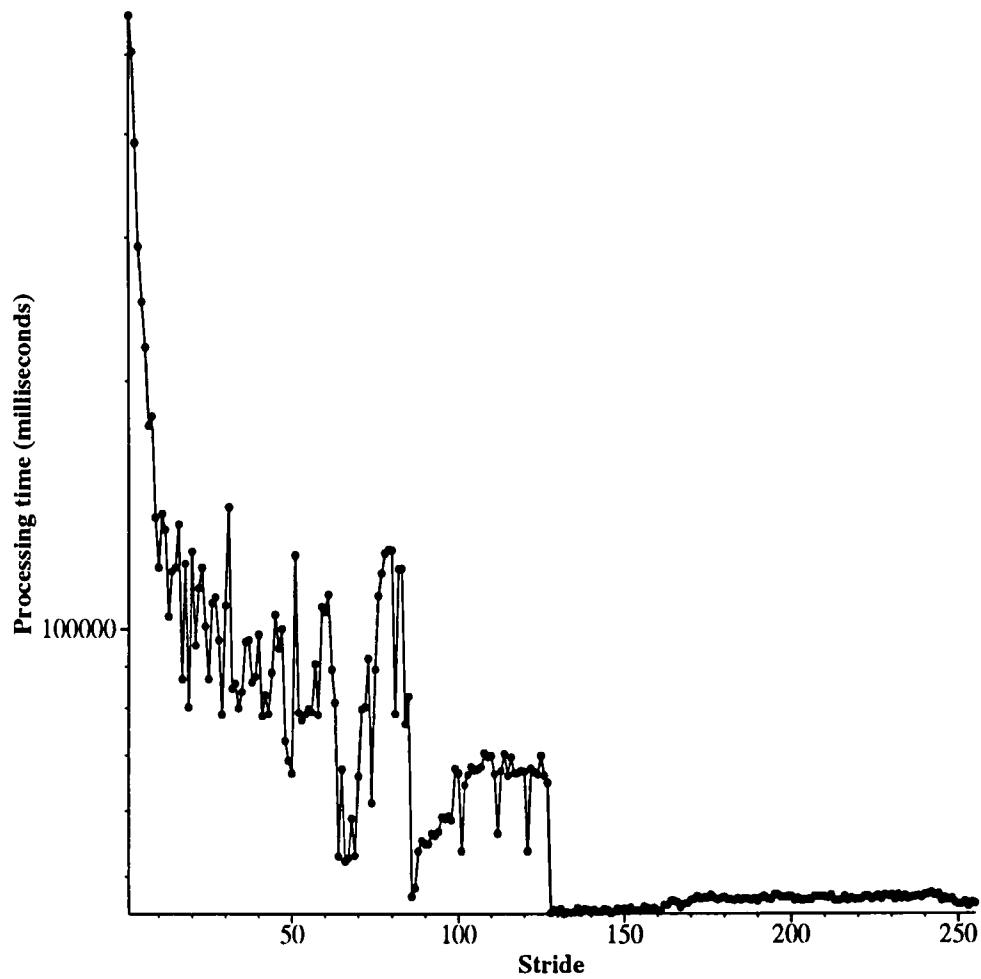


Figure 4.7: Total elapsed time of dendrone construction for the image in Figure 4.2 using the non-recursive Connectivity Filling algorithm.

Table 4.11: Elapsed times of dendrone construction (at selected stride values) for the image in Figure 4.2 using the non-recursive Connectivity Filling algorithm.

Stride	Pixel Sorting	Segmentation	Merging	Indexing	Object Sorting	Total
1	45392.06	2135.81	510723.41	116.25	810.12	559189.12
5	44930.38	1865.48	199120.08	77.61	4436.14	250433.08
10	45300.02	1549.29	68713.97	37.88	3234.05	118837.15
20	45325.69	1348.69	68755.80	18.26	8754.45	124204.09
30	45420.70	1261.07	59283.42	11.37	853.52	106831.12
50	45388.91	1280.89	19255.86	4.69	561.43	66492.57
100	44932.12	1153.75	12030.96	0.64	8412.92	66531.02
150	45394.00	246.09	0.00	1.20	145.57	45787.54
200	45363.00	1193.44	0.00	0.11	759.59	47316.70
255	45173.09	1269.58	0.00	0.03	0.03	46443.27

Table 4.12: Percentage time for different steps of dendrone construction (at selected stride values) for the image in Figure 4.2 using the non-recursive Connectivity Filling algorithm.

Stride	Pixel Sorting	Segmentation	Merging	Indexing	Object Sorting
1	8.12	0.38	91.33	0.02	0.14
5	17.94	0.74	79.51	0.03	1.77
10	38.12	1.30	57.82	0.03	2.72
20	36.49	1.09	55.36	0.01	7.05
30	42.52	1.18	55.49	0.01	0.80
50	68.26	1.93	28.96	0.01	0.84
100	67.54	1.73	18.08	0.00	12.65
150	99.14	0.54	0.00	0.00	0.32
200	95.87	2.52	0.00	0.00	1.61
255	97.27	2.73	0.00	0.00	0.00

sizes of the objects may be smaller than those when the stride value is larger. As the number of objects increases, the time for building the dendrone increases dramatically. In fact, Figures 4.3 through 4.7 demonstrate that the total dendrone construction time increases (regardless of the segmentation algorithm) as the stride decreases. From the breakdown tables of percentage time for different steps, it is clear which specific step is taking the most time. In the pixel labeling version, dendrone construction time is dominated by the time spent in the object merging step when the stride value is not very large (see Tables 4.4 and 4.10). The object merging step occupies more than 80 percent of the total time. On the contrary, in the Connectivity Filling versions, the pixel sorting process takes a constant time, or over 80 percent of the total time, when the stride is large (see Tables 4.6, 4.8, and 4.12). As the stride decreases, the object merging step occupies larger part of the total time, which is similar to the situation in the pixel labeling version. From the breakdown tables of elapsed time (Tables 4.3, 4.5, 4.7, 4.9, and 4.11), it can be seen that at most stride values, the total construction times for three versions are within the same magnitude. However, the pixel labeling algorithm is a little slower than the Connectivity Filling versions in only a few cases. For example, for input image Figure 4.1, when the stride value is 50, the pixel labeling version takes about 18,960 milliseconds and the two Connectivity Filling versions take 11,706 and 11,477 milliseconds, respectively. On average, the pixel labeling version is faster (by a factor ranging from 1.02 when the stride value is 1, to 14.5 when the stride value is 255) than the other versions.

Another aspect of the complexity is space. While all three versions require a tremendous amount of memory (illustrated in Table 4.13), the pixel labeling version excels in that it does not require any memory beyond that needed to store the dendrone itself. Both the recursive and non-recursive Connectivity Filling versions must use auxiliary memory to store the sorted pixels. In addition, the recursive Connectivity Fill version relies on the operating system for recursive function invocations. It is quite possible that the size and contents of an input image will cause the system's internal stack to overflow.

Table 4.13: Memory usage of dendrone construction (stride = 30).

Figure	Segmentation Algorithm	Process Virtual Memory Size (kilobytes)
4.1	Pixel Labeling	2048
4.1	Recursive Connectivity Filling	2536
4.1	Non-recursive Connectivity Filling	2328
4.2	Pixel Labeling	3688
4.2	Recursive Connectivity Filling	4984

## 4.2 Effectiveness Evaluation

The effectiveness of the object matching algorithm using distance-to-centroid image signatures (see Section 2.2.2) was evaluated using both artificially generated images and real-world complex images.

A number of different images have been used to evaluate the effectiveness of the

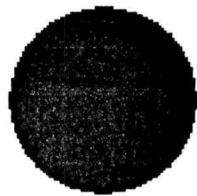
distance-to-centroid image signature matching algorithm. Table 4.14 and Figure 4.8 illustrate four of the images used for the evaluation.

Table 4.14: Summary of images used to evaluate the effectiveness of the distance-to-centroid image signature matching algorithm.

Figure	Type	Description
4.8(a)	artificial	circle object
4.8(b)	artificial	objects with different shapes
4.8(c)	photograph	747 airplane extracted from Figure 4.1
4.8(d)	photograph	another high-altitude view of the 747 airplane from Figure 4.1 (later in time)

#### 4.2.1 Matching Artificial Shapes

Theoretically, the distance-to-centroid image signature is invariant to translation, rotation, and scaling (see [Rau94]). To test these properties, two artificially generated images are used. The images are binary, which means that there are only two intensity values in the images and any pixel is either black (with an intensity value of 0) or white (with an intensity value of 255). The shape for matching (or query shape) is a circle (illustrated in Figure 4.8(a)) and the shapes to match are in a composite image (illustrated in Figure 4.8(b)) of different shapes such as circles, rectangles, triangles, polygons, and objects with holes. The matching algorithm should match the circles best, although they are not of the same size as the source circle, then shapes similar to the circle, and finally the other shapes. Figure 4.9 illustrates the matched shapes from the image in Figure 4.8(b) according to the value of  $S$  from Equation (2.3), in order



(a)



(b)



(c)



(d)

Figure 4.8: Images used to evaluate the effectiveness of the distance-to-centroid image signature matching algorithm.








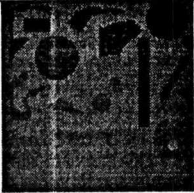








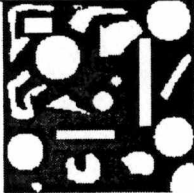








 (1: 97.85%)	 (2: 97.59%)	 (3: 95.22%)	 (4: 91.62%)	 (5: 88.88%)
 (6: 86.25%)	 (7: 85.93%)	 (8: 75.34%)	 (9: 75.00%)	 (10: 70.96%)
 (11: 65.74%)	 (12: 63.28%)	 (13: 66.02%)	 (14: 60.00%)	 (15: 58.11%)
 (16: 55.26%)	 (17: 55.00%)	 (18: 52.05%)	 (19: 50.20%)	 (20: 44.16%)
 (21: 43.44%)	 (22: 42.95%)	 (23: 37.69%)	 (24: 0.00%)	

Figure 4.9: Matching shapes from Figure 4.8(b).

of percent similarity (shown in parentheses). The first object is judged most similar while the object in frame 24 is the least similar. From Figure 4.9, it is clear that the best matches are circles, then some irregular shapes, and most of the long and thin rectangles are ranked last. It can be noticed that the eleventh match is a circle with a hole in it. This match is not ranked very high although the overall shape is a circle. However, the signature image of this object contains two curves. One corresponds to the edge of the larger circle and the other corresponds to the edge of the inner hole. As calculated by Equation (2.3), the similarity value  $S$  is not only determined by the area between the curve corresponding to the large circle and the curve in the source circle signature image, but also by the area between the curve corresponding to the inner hole and the curve in the source circle signature image. As a result, the value of  $S$  in this case is much smaller so that this object is not ranked as high as the solid circles.

#### 4.2.2 Matching Objects from Real Images

Tests using artificially generated images that contain simple shapes have proven that the distance-to-centroid image signature algorithm is effective theoretically. To further test its effectiveness on real-world (more complex) images, in which the shapes and intensity values of objects are much more diverse and irregular, two aerial photographs of Boeing 747 airplanes were used. The goal in this case is to locate the plane in the target image (illustrated in Figure 4.8(d)), which contains not only the plane, but also other background objects. The query image contains only the plane (illustrated in Figure 4.8(c)), which is a reconstructed object image (from the image in Figure 4.1) obtained

via sub-dendrone extraction.

Compared with simple shape matching, matching objects in real-world complex images involves more trial and error. In such images, small objects that contain only a few pixels are much more likely to be present. Hence, the minimum and maximum object sizes for matching can be used to filter out very small objects during the matching process. Due to the size of the object, a small object's distance-to-centroid signature often only contains a few pixels, which is not very useful in the computation of the signature image. In addition, when a larger object is compared with a very small object, the larger object's signature image is scaled to be the same size of the smaller object's signature image, which may cause the loss of some important features in the scaled signature image. In the extreme case, if the smaller object has only one pixel, its signature image will also have one pixel only. Even if the larger object's signature image has many features, the scaled signature image will have only one pixel and the matching algorithm will recognize the two images as a perfect match.

In real-world images, intensity values also play an important role in the matching. By combining shape matching and object surface structure matching, better results may be achieved. Figure 4.10 illustrates the top twelve matched objects from the image in Figure 4.8(d) according to the similarity value  $S$  listed in Equation 2.3. During the matching, only objects whose sizes are between 30 and 60 pixels were processed and only object shapes were matched using Equation (2.3). The best match is the plane plus some background noise near the wings. Therefore, the distance-to-centroid image

(1: 87.20%)	(2: 85.83%)	(3: 84.89%)	(4: 84.06%)
(5: 83.78%)	(6: 83.65%)	(7: 83.26%)	(8: 82.90%)
(9: 82.42%)	(10: 82.18%)	(11: 81.38%)	(12: 81.19%)

Figure 4.10: Matching objects from Figure 4.8(d).

signature matching algorithm also works well for this particular real-world image. Of course, the effectiveness of this algorithm may well depend on the contents of the input image.

Another very important issue regarding object matching is that since the matching is based on (sub-)dendrones, the construction of the dendrone from the input image, particularly the choice of the stride value, is crucial to the effectiveness of the matching process. If the stride value is too large, some objects may never be presented in the dendrone and consequently will never be matched. On the other hand, if the stride value is too small, too many objects that are very similar (only different by a few pixels) will be present in the dendrones, which not only slows down the matching process, but also makes the matching results difficult to differentiate.

## Chapter 5

# Conclusions

Dendronic image characterization is a data-driven and self-structuring process, which can be used in application areas such as military surveillance, medical image analysis, and computer graphics. The algorithm for dendrone construction is robust and strikes a balance between computational time and memory consumption. The algorithm for matching sub-dendrones according to distance-to-centroid signatures is also simple and effective.

**DICE** is a flexible, extendable, and portable implementation of dendronic image characterization. The dendrone library and API's allow users to easily extend DICE's functionalities and add alternative algorithms. The graphical user interface provides a simple and straightforward way to analyze images using their dendronic signatures and locate objects with similar shapes (via sub-dendrones) from multiple images.

Currently, a serial implementation for dendrone construction is used within DICE.

Certainly an optimized parallel approach to image segmentation would speed up the object merging process.

Research in shape matching would provide insight into what alternative object matching algorithm could be used for locating objects with similar shapes. Also, annotating a sub-dendrone with text describing the corresponding objects could achieve better results for information retrieval purposes.

Aside from algorithms, the DICE software environment could be extended in a few functional areas. Encoding dendrones could not only decrease memory usage but also achieve better performance. A tool for annotating images would be desirable. The abilities to maintain a large collection of images and their dendronic signatures would also be desirable.

# Bibliography

# Bibliography

- [CW98] Mary Campione and Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.
- [Eck98] Bruce Eckel. *Thinking in Java*. Prentice Hall, Englewood Cliffs, New Jersey, 1998.
- [FSN+95] Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by Image and Video Content: The QBIC System. *Computer*, 28(9):18–22, 1995.
- [GR95] Venkat N. Gudivada and Vijay V. Raghavan. Content-Based Image Retrieval Systems. *Computer*, 28(9):18–22, 1995.
- [GW87] R. C. Gonzalez and P. Wintz. *Digital Image Processing*. Addison-Wesley, Reading, Massachusetts, second edition, 1987.



- [Har97] William W. Hargrove. Use of Dendronal Signatures to Identify Common Targets in Multiple Remote Images. Technical report, Oak Ridge National Laboratory, Oak Ridge, Tennessee, January 1997.
- [HN91] Arnaud Le Hors and Golas Nahaboo. *XPM The X PixMap Format*. BULL Research FRANCE - Sophia Antipolis, 1991.
- [Jai89] Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [OS95] Virginia E. Ogle and Michael Stonebraker. Chabot: Retrieval from a Relational Database of Images. *Computer*, 28(9):18-22, 1995.
- [Pav82] Theodosios Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville, Maryland, 1982.
- [Pla] Jim Plank. *Jgraph - Filter for Graph Plotting to Postscript*.
- [Pos91] Jef Poskanzer. *PPM - Portable Pixmap File Format*, 1991.
- [Rau94] Thomas W. Rauber. Two-Dimensional Shape Description. Technical Report GR UNINOVA-RT-10-94, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Lisboa, Portugal, 1994.
- [ZS84] T. Y. Zhang and C. Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. *Communications of The ACM*, 27(3):236-239, 1984.

## Appendix

# API Tables of C++ Classes in DICE

Table A.1: API methods of the *Image* class.

Method	Description
Image()	creates an empty <i>Image</i> object (without allocating memory for pixels)
Image(int rows, int columns)	creates an <i>Image</i> object with allocated memory for <i>rows</i> × <i>columns</i> pixels)
~Image()	destroys the <i>Image</i> object
int getRows()	returns the number of rows in the image
int getCols()	returns the number of columns in the image
int **getData()	returns a pointer to the image data (in two-dimensional representation)
int *getArray()	returns a pointer to the image data (in one-dimensional representation)
void fill( ObjectPtr objectPtr, int **objectData)	paints pixels (pixel coordinates available from the object pointed by <i>objectPtr</i> ) of the image with intensity values (available from <i>objectData</i> )
void fill( ObjectPtr objectPtr, int intensity)	paints pixels (pixel coordinates available from the object pointed by <i>objectPtr</i> ) of the image with <i>intensity</i>
void fill( ObjectPtr objectPtr, long id)	paints pixels (pixel coordinates available from the object pointed by <i>objectPtr</i> ) of the image with <i>id</i>
void writeToFile(char *fileName)	writes the <i>Image</i> object to a file in PPM format
void view( char *fileName, char *imageViewPath)	displays the image file

Table A.2: API methods of the *XpmImage* class.

Method	Description
XpmImage(char *fileName)	creates an <i>XpmImage</i> object from a file
~XpmImage()	destroys the <i>XpmImage</i> object

Table A.3: API methods of the *Histogram* class.

Method	Description
Histogram(Image *image)	creates a <i>Histogram</i> object from an <i>Image</i> object
void display()	displays the histogram using an image viewer chosen by the user ( <i>xv</i> , which is an interactive image displayer for the X Window system, by default)
void saveToPSFile(char *fileName)	saves the histogram to a file in PostScript format
void deletePSFile()	deletes the saved PostScript file

Table A.4: API methods of the *Option* class (part 1).

Method	Description
<code>Option(int argc, char **argv)</code>	creates an <i>Option</i> object from command line arguments
<code>Option(char *fileName, char *rootFileName, int stride, bool fullDump,  bool noDendrogram,  bool displayID,  bool reverse)</code>	creates an <i>Option</i> object from parameters: input image file name output image file name stride value to segment the image <i>true</i> (1) if pixel information should be generated and <i>false</i> (0) otherwise <i>true</i> (1) if no dendrogram should be generated and <i>false</i> (0) otherwise <i>true</i> (1) if composite object ids should be generated in the dendrogram and <i>false</i> (0) otherwise <i>true</i> (1) if the image should be segmented from the lowest intensity value to the highest one and <i>false</i> (0) otherwise
<code>~Option()</code>	destroys the <i>Option</i> object
<code>char *getImageFileName()</code>	returns the input image file name
<code>XpmImage *getImage()</code>	returns a pointer to the <i>XpmImage</i> object
<code>int getLeft()</code>	returns the left <i>x</i> -coordinate of the sub-image from which to generate dendrones
<code>int getTop()</code>	returns the top <i>y</i> -coordinate of the sub-image from which to generate dendrones
<code>int getRight()</code>	returns the right <i>x</i> -coordinate of the sub-image from which to generate dendrones
<code>int getBottom()</code>	returns the bottom <i>y</i> -coordinate of the sub-image from which to generate dendrones
<code>int getStride()</code>	returns the stride value to segment the image
<code>char *getRootFileName()</code>	returns the root file name from which generated output file names are formed

Table A.5: API methods of the *Option* class (part 2).

Method	Description
bool getFullDump()	returns <i>true</i> (1) if the generated dendrones contain pixel information and <i>false</i> (0) otherwise
bool getNoDendrogram()	returns <i>true</i> (1) if the no PostScript dendrograms should be generated and <i>false</i> (0) otherwise
bool getDisplayID()	returns <i>true</i> (1) if the no composite object ids should be used in the dendrograms and <i>false</i> (0) otherwise
bool getReverse()	returns <i>true</i> (1) if the image is segmented from the lowest to the highest intensity value and <i>false</i> (0) otherwise

Table A.6: API methods of the *Object* class (part 1).

Method	Description
Object(int row, int col)	creates an <i>Object</i> object containing one pixel ( <i>row</i> , <i>col</i> )
Object(FILE *fp, Object *parentObjectPtr, int **data)	creates an <i>Object</i> object from a file
Object(Object *subObject1Ptr, Object *subObject2Ptr)	creates an <i>Object</i> object as the parent object of two sub-objects
~Object()	destroys the <i>Object</i> object
long getID()	returns the id of the object
int getStartRow()	returns the top <i>y</i> -coordinate of the object's bounding box
int getStartCol()	returns the left <i>x</i> -coordinate of the object's bounding box
int getEndRow()	returns the bottom <i>y</i> -coordinate of the object's bounding box
int getEndCol()	returns the right <i>x</i> -coordinate of the object's bounding box
float getCenterRow()	returns the center <i>y</i> -coordinate of the object's bounding box
float getCenterCol()	returns the center <i>x</i> -coordinate of the object's bounding box
long getSize()	returns the size (in terms of number of pixels) of the object
int getChildrenNumber()	returns the number of sub-objects
Object **getChildren()	returns a pointer to an array containing the sub-objects
Object *getChild(int i)	returns a pointer to the <i>i</i> th sub-object
Object *getParent()	returns a pointer to the parent object
long getDescendentNumber()	returns the number of descendent objects of the object (including the object itself)
long getMaxObjectID()	returns the maximum object id in the sub-dendrone
Pixel *getPixel(long i)	returns a pointer to the <i>i</i> th pixel contained in the object

Table A.7: API methods of the *Object* class (part 2).

Method	Description
Pixel *getAllPixels()	returns a pointer to an array of pixels contained in the object including those in its sub-objects
bool isComposite()	returns <i>true</i> (1) if the object is composite and <i>false</i> (0) otherwise
bool isEmpty()	returns <i>true</i> (1) if the object is empty and <i>false</i> (0) otherwise
void setID(long id)	sets the object id
void setStartRow(int startRow)	sets the top <i>y</i> -coordinate of the object's bounding box
void setStartCol(int startCol)	sets the left <i>x</i> -coordinate of the object's bounding box
void setEndRow(int endRow)	sets the bottom <i>y</i> -coordinate of the object's bounding box
void setEndCol(int endCol)	sets the right <i>x</i> -coordinate of the object's bounding box
void setCenter()	sets the center <i>x</i> -coordinate and <i>y</i> -coordinate of the object's bounding box
void setChild(long i, Object *objectPtr)	sets the object pointed by <i>objectPtr</i> as the <i>i</i> th sub-object
void addChild(Object *objectPtr)	adds a sub-object
void swapChildrenPosition(int i, int j)	swaps positions of the <i>i</i> th and <i>j</i> th sub-objects
void mergeParents(Object *objectPtr)	merges the parent objects into one
void addPixel(int y, int x)	adds pixel ( <i>x</i> , <i>y</i> ) to the object
void mergePixels(Object *objectPtr)	merges pixels of two objects
bool touch(Object *objectPtr)	returns <i>true</i> (1) if the two objects touch and <i>false</i> (0) otherwise
void printFull(FILE *fp)	dumps full object information into a file
void printPartial(FILE *fp)	dumps partial object information into a file
void printPixels(FILE *fp, int **data)	dumps pixel information into a file
bool checkSelf(FILE *fp)	checks the object integrity
float similarity(Object *objectPtr)	returns the similarity value (with respect to surface structure) between two objects



Table A.8: API methods of the *Dendrone* class (part 1).

Method	Description
Dendrone( Image *image, int left, int top, int right, int bottom, int stride, bool reverse)	creates an empty <i>Dendrone</i> object from an image
Dendrone(char *fileName)	creates a <i>Dendrone</i> object from a file
~Dendrone()	destroys the <i>Dendrone</i> object
int getStride()	returns the stride value to segment the image
int getLeft()	returns the left <i>x</i> -coordinate of the sub-image from which to generate dendrones
int getTop()	returns the top <i>y</i> -coordinate of the sub-image from which to generate dendrones
int getRight()	returns the right <i>x</i> -coordinate of the sub-image from which to generate dendrones
int getBottom()	returns the bottom <i>y</i> -coordinate of the sub-image from which to generate dendrones
bool getReverse()	returns <i>true</i> (1) if the image is segmented from the lowest intensity value to the highest one and <i>false</i> (0) otherwise
long getObjectNumber()	returns the maximum possible number of objects in the dendrone
long getRealObjectNumber()	returns the exact number of objects in the dendrone
ObjectPtr getObject(long i)	returns a pointer to object <i>i</i> in the dendrone
ObjectPtr getRootObject()	returns the root object in the dendrone
long getParentObjectID(long i)	returns id of the parent object of object <i>i</i>
int getChildrenNumber(long i)	returns number of sub-objects of object <i>i</i>
long getChildObjectID( long i, int j)	returns id of the <i>j</i> th sub-object of object <i>i</i>
long getMinObjectSize()	returns size of the smallest object in the dendrone
long getMaxObjectSize()	returns size of the largest object in the dendrone

Table A.9: API methods of the *Dendrone* class (part 2).

Method	Description
void buildDendronalTree()	constructs the dendrone
void displayDendronalTree( char *rootFileName, long objectID, bool fullDump, char *dendroneFileName)	dumps the sub-dendrone with object <i>objectID</i> as the root to a file
void freeDendronalTree( ObjectPtr objectPtr)	frees memory of the dendronal tree
long searchPixel(int x, int y)	returns the id of the smallest object containing pixel ( <i>x</i> , <i>y</i> )
void generateNCDendrogram( long id, int minGrayLevel, int maxGrayLevel, bool displayID, char *rootFileName, char *psFileName)	generates a non-coordinate dendrogram
void generateXDendrogram( long id, int minGrayLevel, int maxGrayLevel, bool displayID, bool wholeTree, char *rootFileName, char *psFileName)	generates an <i>x</i> -coordinate dendrogram
void generateYDendrogram( long id, int minGrayLevel, int maxGrayLevel, bool displayID, bool wholeTree, char *rootFileName, char *psFileName)	generates a <i>y</i> -coordinate dendrogram
void viewDendrogram(char *psFileName)	displays the dendrogram

Table A.10: API methods of the *Dendrone* class (part 3).

Method	Description
Image *reconstructImage( int threshold, char *rootFileName, char *fileName)	reconstructs an image containing objects whose intensities are greater than <i>threshold</i>
Image *reconstructImage( long id, long fillID, char *rootFileName, char *fileName)	reconstructs an image of object <i>id</i>
bool readDendronalTreeFromFile( char *fileName)	reads a dendrone from a file and return <i>true</i> (1) if successful and <i>false</i> (0) otherwise

Table A.11: API methods of the *Shape* class.

Method	Description
Shape(ObjectPtr objectPtr)	creates a <i>Shape</i> (distance-to-centroid signature) object
~Shape()	destroys the <i>Shape</i> object
float similarityByPoint(Shape *shapePtr)	returns the similarity of two objects according to Equation (2.2)
float similarityByArea(Shape *shapePtr)	returns the similarity of two objects according to Equation (2.3)

Table A.12: API methods of the *RankList* class.

Method	Description
<code>RankList()</code>	creates an empty <i>RankList</i> object
<code>~RankList()</code>	destroys the <i>RankList</i> object
<code>void append(long id, float score, long size)</code>	appends an item to the ranked list
<code>void sortByScore()</code>	sorts the ranked list by score in descending order
<code>long getItemNumber()</code>	returns the number of items in the ranked list
<code>long getObjectID(long i)</code>	returns the <i>i</i> th item's <i>objectID</i> field
<code>float getScore(long i)</code>	returns the <i>i</i> th item's <i>score</i> field
<code>long getSize(long i)</code>	returns the <i>i</i> th item's <i>size</i> field
<code>void display()</code>	displays the ranked list in standard output

## Vita

Luojian Chen received a Bachelor of Science degree in Computer Science in 1994 from Shanghai Jiaotong University in Shanghai, China. In August 1996, he entered the Computer Science graduate program at the University of Tennessee in Knoxville, Tennessee and received a Master of Science degree in Computer Science in May 1998.