University of Alberta

Image Databases:

A Content-Based Type System

and

Query By Similarity Match

by

Irene Lin-Oi, Cheng

Technical Report TR 99-03

May 1999

DEPARTMENT OF COMPUTING SCIENCE

University of Alberta

Edmonton, Alberta, Canada

Abstract

The use of on-line image repositories is growing and becoming commonplace. Due to the inadequacy of traditional databases in handling complex images and voluminous data, new designs and techniques are needed to efficiently organize, store, manage and retrieve images. It is felt that the Structural Query Language (SQL) and Object Query Language (OQL) lack the expressive power to describe image queries. Recently the Multimedia Object Query Language (MOQL) which is an extended version of OQL was defined in a PhD thesis at the University of Alberta. As part of the DISIMA (Distributed Image Database Management System) project, one goal of this report is to design and implement a content-based generic type system to support the storage and retrieval of images. The other goal is to design and implement a query parser and engine for the MOQL extension.

There has been research conducted in this area but most of it is application specific, focusing on selected features. The type system, discussed in this report, integrates all the features into a framework, which can be customized to meet the specific needs of applications.

Contents

1 Introduction **5**

1.1 Motivation . . . 5
1.2 Objectives . . . 6
1.3 Research Context — DISIMA System . . . 6
1.4 Scope . . . 9
1.4.1 Type System . . . 9
1.4.2 Query System . . . 10
1.4.3 Why Object-Orientation ? . . . 10
1.5 Contributions . . . 11
1.6 Report Organization . . . 11

2 Related Work **13**

2.1 Image Database Models . . . 13
2.2 DISIMA vs. Other Models . . . 14

3 DISIMA Kernel **16**

3.1 Terminology . . . 16
3.1.1 Logical Salient Object (LSO) . . . 16
3.1.2 Physical Salient Object (PSO) . . . 16
3.2 A Content-Based Generic Type System . . . 16
3.2.1 The Type System Overview . . . 17
3.2.2 Logical Salient Object (LSO) and Physical Salient Object (PSO) . . . 18
3.2.3 Spatial Relationship . . . 18
3.2.4 The Color Feature . . . 24
3.2.5 Geometric Object Hierarchy (The Shape Feature) . . . 25
3.2.6 The Texture Feature . . . 31
3.2.7 Image and Image Representation . . . 32
3.2.8 Methods to support Query Execution . . . 32
3.3 Similarity Match . . . 33
3.3.1 Examples of Similarity Match Algorithms . . . 34

4 Query System **44**

4.1 MOQL Language . . . 44
4.2 A Query Parser for the MOQL Extension . . . 44
4.2.1 New Semantics and Syntax . . . 45
4.2.2 Internal Structure of A Query . . . 48

4.3 A Query Engine for MOQL 49
4.3.1 Relational approach 49
4.3.2 DISIMA approach 49

5 Implementation and Limitations **64**
5.1 Introduction 64
5.1.1 Why ObjectStore? 64
5.1.2 Why Flex and Yacc? 65
5.2 How are the programs organized? 65
5.3 Schema generation 66
5.4 Data population 66
5.5 Database images vs. file system images 67
5.6 Similarity match 68

6 Conclusion **69**
6.1 Contributions 69
6.2 Future Work 69
6.2.1 A full OQL parser and query processor 70
6.2.2 Store images in progressive resolutions 70
6.2.3 Optimization 70
6.2.4 Matching polygons with circles and ellipses 72
6.2.5 Query on the image hierarchy and image attributes 72
6.2.6 Information about the image 72
6.2.7 Database update 72
6.2.8 Replace the grade with a structure 72
6.2.9 Query by sketch or example 72
6.2.10 Extend to 3D still image 72
6.2.11 The video extension 73

Bibliography **74**

A An extraction of the code on I_class and C_class **76**

B An extraction of the parser rules **79**

C An example of the user defined file *userClasses* 82

D Glossary **86**

List of Figures

1.1 DISIMA Architecture . . . 7
1.2 The text of a MOQL query . . . 8
1.3 Graphical Interface Startup Window . . . 9

3.1 Content-Based Generic Type System . . . 17
3.2 Each LSO can be associated with more than one PSO . . . 18
3.3 An example of the LSO class hierarchy . . . 18
3.4 Topological relations . . . 19
3.5 Mbbs may not reflect true topology . . . 20
3.6 Inconsistency between Mbbs and objects topology . . . 20
3.7 An example of using Mbbs to measure direction, i.e., q is east of p . . . 21
3.8 Mbb defined by intervals . . . 22
3.9 How x-intervals are compared . . . 23
3.10 Edge detection may not be accurate . . . 23
3.11 The red, green, blue components of a color . . . 24
3.12 Uses polygon to describe arbitrary shape . . . 25
3.13 The Geometric Object Hierarchy (Logical design) . . . 26
3.14 The Atomic Hierarchy (Flat hierarchy) . . . 27
3.15 The Geometric Object Hierarchy (Final design) . . . 27
3.16 An example of deep search . . . 28
3.17 Orientation of shapes . . . 29
3.18 Examples of composite shapes . . . 29
3.19 An example of texture matching . . . 31
3.20 Shapes equal or similar to a triangle . . . 33
3.21 Boundary represented by chain code . . . 35
3.22 The signature algorithm . . . 35
3.23 Shapes of different scales . . . 36
3.24 Problem with the signature algorithm . . . 36
3.25 An example of turning angle . . . 37
3.26 Polygon with concave angle . . . 37
3.27 Small variation in shape . . . 38
3.28 Implemented version of the turning angle algorithm . . . 39
3.29 Polygon similar to an ellipse . . . 39
3.30 A three stages algorithm to compare composite shapes . . . 39
3.31 Spatial comparison on composite shapes . . . 40
3.32 Histograms containing 5 colors and 1 color . . . 41
3.33 Experiment result using average color . . . 42
3.34 Match spatial relationship based on centroid . . . 43

4.1 An example of a query object 48
4.2 Examples of query trees 50
4.3 A query with a lot of brackets 51
4.4 A nested query 51
4.5 Relational entities and relationships 52
4.6 A relational execution tree 52
4.7 Navigation from the PSO object 53
4.8 DISIMA execution tree 54
4.9 The result of a condition: The Resultpso and Unitpso objects 55
4.10 To process "contains" as a join 56
4.11 The result of the *and* operator 57
4.12 The result of the *or* operator 57
4.13 Temporary result illustrated in disjunctive normal form 60
4.14 Examples of temporary result 61

5.1 The DISIMA directories 65
5.2 An image containing five golden fish as salient objects 66
5.3 The shape of starfish 68

6.1 An example of a query represented by different tree structures 71

Chapter 1

Introduction

1.1 Motivation

Image databases have been gaining attention in recent years because many industrial applications including teleconferencing, commerce, education and medicine require the management of large volumes of images. Database management systems (DBMSs) can be used in this capacity, and there have been a number of initiatives in this direction. However, further research on enhancing existing DBMSs is necessary due to their inefficiency in handling image queries. The inadequancy is due to one or more of the following reasons:

- Keyword matching–
  Some databases support only keyword retrieval. In these databases, images and objects in images are annotated with user-defined keywords. Retrieval of images is based on keyword matching. However, the assignment of keywords is subjective, and the choice of words can vary from person to person. A circular object may be viewed as a balloon by one person, and as a ball by another. It may happen that, the keyword "balloon" is stored in the database, and "ball" is used in the query. As a result, the image annotated as "balloon" will not be retrieved. The other disadvantage of using keyword matching is that, *image content,* such as color, shape and texture, cannot be precisely described using keywords. An efficient image DBMS should therefore support *content-based* matching, both in terms of modeling and in terms of querying.

- Exact matching–
  Image queries on color, shape and texture require *similarity match* between database objects and query predicates, but some databases support only exact match. For example, in the query:
  ```
  SELECT p.salary
  FROM person p
  WHERE p.name = 'Happy';
  ```
  a person's name can be either 'Happy' or not 'Happy'. The query does not retrieve the salary of a person named 'Happier' although the first four characters are the same. This kind of matching is called *exact match.* However, in image databases similarity match is generally preferred. For example, the query:
  ```
  SELECT m
  FROM image m, object o
  WHERE m contains o
  AND o.color similar colorgroup(255,0,0);
  ```
  requires images containing an object which is similar to the color red, corresponding to the red, green and blue values (255,0,0). In the database there may be images containing objects of different

degrees of red: dark red, light red, pink, etc. The idea of similarity match is to have objects, of different degrees of red, retrieved and presented to the user. Images presented are in decreasing similarity order. An important aspect in image databases is the *distance function* or *similarity function* which compares the database object with the target object and assigns a similarity grade between 0 to 1. An exact match is represented by '1'.

- Image-feature specific–
  Most database models focus on one, or a few image features, and not the entire set including color, shape, texture and spatial relationship.

- Query language–
  Existing image DBMSs do not have a semantically rich underlying query language, or the query language they use is not expressive enough to handle image queries.

1.2 Objectives

In view of the deficiency of existing DBMSs to handle image queries, the main objective of this report is to design and implement a content-based generic type system which can efficiently support image storage and retrieval. A secondary objective is to implement a query parser and engine to process queries, in particular MOQL (Multimedia Object Query Language) queries, and return the resulting images in similarity order.

To perform similarity match, some image DBMSs apply real-time image processing techniques. Since images are often complex and voluminous, the content-based type system discussed in this report uses pre-processing in order to reduce the on-line response time. Image features essential for query execution are extracted during the pre-processing stage and stored in the database. Pre-processing is important, especially in a distributed environment such as the Internet, where databases can be located at different sites. Real-time processing of images requires the transmission of gigabytes of raw data to the processing site while pre-processing requires only the transmission of some pre-extracted data.

In order to support image queries efficiently, the DISIMA model adopts the following:

- Content-based matching, in addition to keyword matching.

- Similarity match, in addition to exact match.

- A generic type system, which includes the whole set of image features, and at the same time, allows applications to customize these features according to application needs. The type system also allows applications to expand the logical salient object class hierarchy by defining subclasses, such as Person, Athlete, Building, etc.

- The semantically rich MOQL, which is extended from OQL (Object Query Language), as the underlying query language to handle image queries.

1.3 Research Context — DISIMA System

The report is part of the DISIMA (Distributed Image Database Management System) Project carried out by the Database Research Group at the University of Alberta. DISIMA is funded by The National Science and Engineering Research Council (NSERC) of Canada. The DISIMA architecture is comprised of a number of major components, as shown in Fig.1.1. The focus of Phase I development is on the *User Interface (Visual MOQL), Type System* and *Query Processor.* The *Visual MOQL* [23] component provides a user friendly graphical query interface. The *MOQL* component defines the underlying query language [18] generated by the interface. The *Query Processor* handles the parsing and execution of the generated query. The *Type System* defines the data structures and methods which support database population and image retrieval by similarity match.

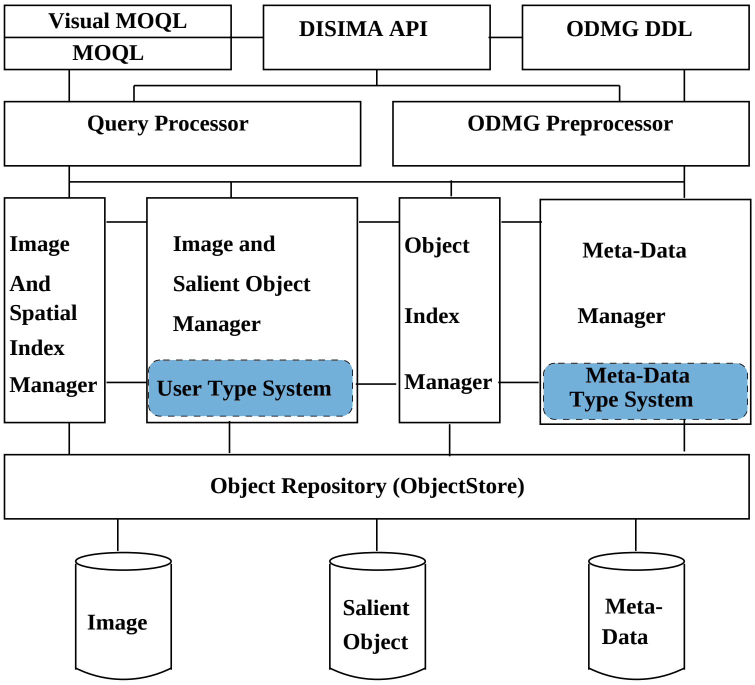


Figure 1.1: DISIMA Architecture

DISIMA is built on top of object repositories (ObjectStore is used as a repository in the current prototype). The *Image and Spatial Index Manager,* and *Object Index Manager* are included in the DISIMA architecture, because the object repositories may not have indices, and even if they do, their indices may not meet the DISIMA requirements. Moreover, these Index Managers need to dynamically integrate new indexes. Meta-information about images and objects is handled by the *Meta-Data Manager.* Meta-data is a kind of important on-line data to ensure the availability and quality of the information delivered.

Visual query language is expected to be more popular in query language evolution, because it provides a user friendly interface that allows easy composition of queries. Since images are inherently visual, users would prefer to use a graphical language instead of typing the query string. An example will help to understand how the DISIMA type system and query system work with the graphical interface. Suppose the image query is: "Find images with 2 people next to each other without any building, or images with buildings without people, or images with animals". Figure 1.2 shows the query written in MOQL. Although the user may know exactly what kind of image he/she wants, typing out the query text without any syntax or semantic error is not straightforward.

A graphical language makes use of pictorial information to represent objects, and the relationships defined among them. In general, the expressive power of graphical languages is low because they often do not have an underlying textual query language. The DISIMA model has its textual query language MOQL, which extends the standard OQL [5] by adding spatial, temporal, and presentation properties for content-based image and video data retrieval, as well as for queries on structured documents. The

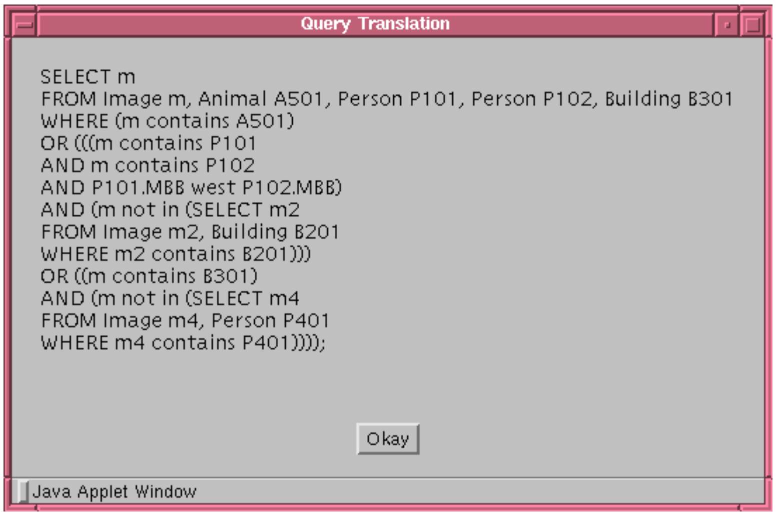


Figure 1.2: The text of a MOQL query

image part of MOQL is implemented in phase I of the DISIMA project. Users can specify image queries through a graphical interface. The query information is then translated into MOQL, and executed by the query system.

The DISIMA graphical interface allows users to query the database by specifying the salient objects in the image. The query can then be refined by defining the color, the shape and other attributes. The spatial relationships among these salient objects can also be specified. The graphical interface startup window (Figure 1.3), consists of a number of components:

- A chooser to select the image classes.
- A salient object class browser to choose the desired objects.
- A horizontal slider to specify the maximum number of images that will be returned.
- A horizontally slider to specify the minimum similarity required from the returned images.
- A working canvas where queries can be constructed.
- A query canvas where the user can compose compound queries, by using the AND, OR and NOT operators.

After the user has clicked the "Query" button on the query canvas, the corresponding MOQL will be generated automatically, and passed to the query system. The MOQL in Figure 1.2 is generated from the query specified in Figure 1.3.

Another important aspect in the DISIMA model is the separation of *logical salient objects* (LSOs) from *physical salient objects* (PSOs). DISIMA views the content of an image as a set of salient objects. Traditional data, i.e., textual descriptions and keywords, are associated with the LSOs while image features such as spatial relationship, color, shape and texture are associated with the PSOs. LSOs and PSOs have a one-to-many relationship which allows the same logical entity to have different physical appearances in images. The detail of LSOs and PSOs will be discussed in Section 3.2.2. The representation of salient objects and the spatial relationships among them assume objects detection. Salient object detection combines manual and automatic interpretation of images. The automatic process applies several pattern recognition and analysis algorithms to capture the content of an image. The manual process complements the automatic one. The current development focuses on face detection,

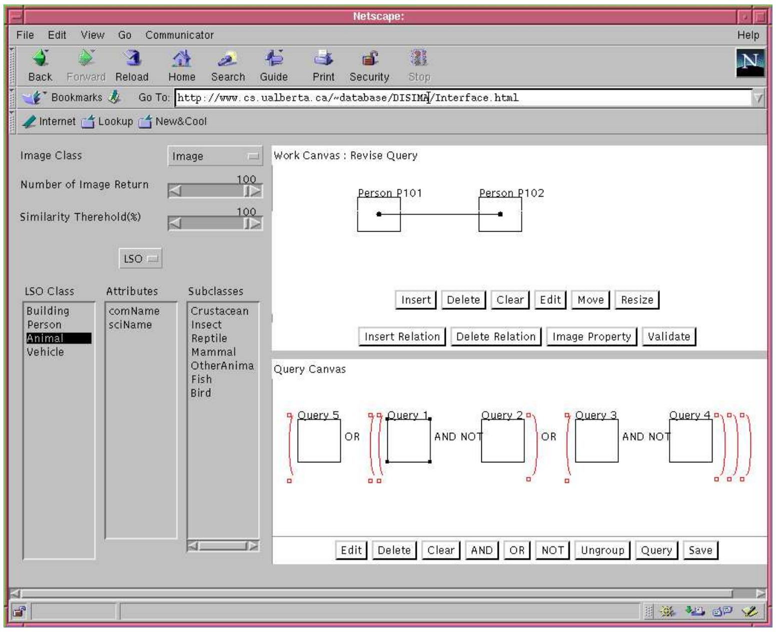


Figure 1.3: Graphical Interface Startup Window

for the reason that the driving application of news image database contains pictures with many person objects.

Images and their representations are also independent in the DISIMA model. The separation allows an image to be represented in different formats, i.e. JPEG, GIF, TIF, etc. The JPEG format is used as the default in the system. How images are related to their representations, and the PSOs will be discussed in Section 3.2.7.

1.4 Scope

The design and implementation described in this report follow the object-oriented (OO) approach, and focus on two main areas: the type system (DISIMA kernel) and the query system. Data population, which involves objects extraction and annotation, is a part of the implementation. However, since the extraction and annotation techniques are related to the research of image processing, their evaluations are not included in this report. The implementation makes use of the techniques currently available, which are not yet fully automatic.

1.4.1 Type System

As mentioned in Section 1.1, modern image DBMSs need to support content-based, rather than keyword-based, search. Therefore, the DISIMA type system is designed to incorporate the image features in addition to keywords. The DISIMA type system has the following characteristics:

- In the DISIMA images, spatial relationship between objects are defined in addition to color, shape and texture.
- Since the shape of an object can vary perceptually depending on the viewing angle and zoom level, the DISIMA type system allows more than one shape to be associated with each object.

- In the DISIMA system, a shape object can be composite or atomic. A composite shape is comprised of more than one atomic shape.
- The type system can support image storage either in the file system or in the database. An image can be displayed in different resolutions, and each resolution can be stored in a different format, i.e. JPEG, TIF, GIF, etc.
- The image features, i.e. color, shape, texture and spatial relationship, are defined independently from each other, allowing an application to use a selected set. For example, a road map application can use the shape and spatial features, and omit the color and texture features.
- In the logical design of the type hierarchy, a subclass is expected to be more specific than its superclass by imposing additional constraints. This can cause conflicts during implementation if the constrained class needs less data element than the non-constrained class. By applying multiple inheritance, the DISIMA type system resolves this conflict. Detailed discussion can be found in Section 3.2.5.

Query methods are implemented in the type system to perform similarity match. These query methods are based on the color, shape, texture and spatial relationship extraction techniques discussed in the image processing literature. Since image processing is a separate research topic, the performance evaluation of these techniques is not addressed.

1.4.2 Query System

The query system is comprised of two components: a parser and an engine.

- Parser– It is not the intention of this report to duplicate the effort of developing a complete parser for the Structural Query Language (SQL) or Object Query Language (OQL). Instead, the focus is on implementing the extension for OQL parsers, in order to handle the new syntax introduced by MOQL. The DISIMA parser performs two tasks: it checks the syntax of the input query, and constructs an internal representation (query object) of the query, which will be executed by the query engine.
- Query Engine– A query object contains all the information specified in the user query. The search conditions defined in the query are stored in a tree structure, inside the query object. By traversing the tree structure, the engine is able to retrieve the qualified images, and assign similarity grades to them. Only images graded higher than the requested similarity threshold are returned to the user.

  A unique feature of the query engine is the data structure used to store intermediate results of a query. The process is similar to image sketching; objects satisfying the search conditions are inserted into the structure, in the same way as objects are sketched onto an image. Objects disqualified by a condition are removed from the structure.

  Similar to the DISIMA parser, the engine focuses on the MOQL extension. Comparison predicates using >, >=, <, <=, =, <>, and the *in* predicate are supported, while other SQL and OQL predicates such as *like, exists, any, all,* and *some,* are not implemented in the current phase of the DISIMA system.

1.4.3 Why Object-Orientation ?

Relational databases have proved quite successful in supporting traditional business applications. However, they are insufficient for more complex applications such as those used in image and graphic databases, geographic information, and multimedia databases. These new applications require more complex data structures, longer-duration transactions, new data types for storing images or large textual items, and the ability to define nonstandard application-specific operations. A key feature of object DBMSs is the power to specify both the structure of complex objects and the operations that can be applied to these objects.

In view of the complex structures, efficient retrieval is one of the main goals in image databases. In relational databases, the entity and relationship tables are in first normal form allowing only atomic attributes. In order to obtain the data of a composite attribute, expensive join-operations are performed. On the other hand, the OO approach supports *associative access* (to find objects with certain properties in the database) and *navigational access* (to further investigate the structure of the found object) based on *object identities (oid),* reducing the number of expensive join-operations. Associative access and navigational access are used by the DISIMA query parser to build the query tree, and by the query engine to execute the query–which will be discussed in Sections 4.2 and 4.3.

Another characteristic of the OO approach is the ability to create classes to organize objects, to create objects, to structure an inheritance hierarchy to organize classes so that subclasses may inherit attributes and methods from superclasses, and to call methods to access specific objects. Inheritance, in particular multiple inheritance, is fully applied in the design of the DISIMA shape hierarchy which will be discussed in Section 3.2.5.

The OO approach has the advantage of code reusability and interoperability. Reusability is partly contributed by inheritance and partly by encapsulation. Encapsulation makes the code more portable and easily adopted by other modules or programs. Encapsulation also facilitates interoperability. By defining the access methods or interface, foreign systems can access the database without worrying about the complex data structures. Interoperability is essential in a distributed environment, especially between heterogeneous databases running on different platforms.

1.5 Contributions

This report has achieved the objective of constructing the framework of a content-based generic type system to support query by similarity match. The main characteristics of the framework are as follow:

- Customization–Applications can incorporate user-defined logical salient object classes, or use a selected set of image features.
- In addition to the spatial relationship concept defined in the MOQL extension, the semantics covering color, shape and texture are also implemented.
- The report introduced the disjunctive normal form concept to process intermediate results for image queries.
- The current query system provides a stepping stone, on which indexing and optimization can be developed.
- Future research work inspired by this report includes three dimensional images, video and query by example.

1.6 Report Organization

This report contains six chapters followed by the Bibliography and Appendix. Chapter 1 is the introduction. Chapter 2 reviews recent image database models in the literature, and compares them with DISIMA. Chapter 3 explains the design of the DISIMA kernel and discusses how similarity match is supported by the system. Chapter 4 explains the design of the query system, and how the parser and engine retrieve images from the database. Chapter 5 describes implementation aspects. Chapters 6 is the conclusion and future enhancements.

Chapter 2

Related Work

Before looking at the DISIMA model, this chapter reviews the different image database models in the literature. These models are designed to serve different application domains.

2.1 Image Database Models

The traditional DBMSs, e.g. Oracle, using SQL as the underlying query language, are based on textual annotation. Images, or objects in images, are associated with keywords which provide information about the images. The corresponding images are retrieved when the associated keywords match the query predicates. Annotation does not rely on image processing techniques, because the keywords are user-defined, and the data structure to store keywords is simple. However, the disadvantage is inconsistency because the assignment of keywords is subjective; different users may use different keywords. Furthermore, the image content–such as color, shape and texture–cannot be represented precisely by keywords.

The Geographical Information System (GIS) model ([6], [20] and [26]) is designed for analyzing geographical information such as distances and relative directions. Examples of its applications are road maps, city maps and satellite images. In this model, predefined symbols, such as a dot, a solid line or a broken line are used to represent the sites of interest (e.g., beach, hotel, river, etc.). Spatial relationships, i.e., north and east, and the distances between symbols are stored in the database. An example query is "*select all the beaches within 1 mile of the Hyatt Hotel*". This model uses geometric elements such as points, lines, regions and vectors, which can normally be defined in geometry. The other image features, such as color and texture, are not of particular importance.

The third model is shape-based. Spatial, color and texture features are not essential. An example application is a trademark database which stores company logos. The database is queried to make sure that no similar pattern is found before a new company can register its logo. This model gives more freedom to the shape feature than the GIS model does, because it allows arbitrary shapes in addition to the basic geometric shapes. However, arbitrary shapes cannot be defined by simple geometric formulae. Additional information about the object boundaries is required and more complicated shape matching algorithms are necessary. This model is discussed in [9] and [28].

The fourth model focuses on the image content (e.g., [8], [3] and [1]). The user can query on any of the color, shape and texture features. An image which matches the requested values will be retrieved and displayed in order of similarity. Different distance functions (similarity functions) have to be defined for the different features. In order to process and retrieve data efficiently, data structures and indices which are more complex than the previous models are required.

The above four models handle still images but not video images. The fifth model includes the temporal feature in order to capture the movements of objects [15]. A video is a collection of frames, each of which can be considered a still images. The temporal feature defines the time slot between frames and thus the movement of objects over time can be computed. This model is useful for video databases.

The work of this report is part of the DISIMA project. The DISIMA model is similar to the fifth model described above, with the addition of the spatial features and distribution capability. The current phase of the project covers only still images. Video and distribution will be developed later. An overview of the DISIMA architecture is given in Section 1.3.

2.2 DISIMA vs. Other Models

The traditional entities-relational database model, using SQL as the underlying query language, supports only keyword and exact matching. The object database model using OQL enhances the expressive power in queries, by allowing function calls in the predicates, but OQL is still inadequate to handle image queries, which requires content-based and similarity matching. The DISIMA model overcomes the inadequacy by supporting content-based and similarity matching, in addition to keyword and exact matching.

Although in recent years, an increased effort has been dedicated to the research of image DBMSs, not all of the models discussed in the literature are comparable with the DISIMA model. The GIS applications ([6], [20] and [26]) deal specifically with geographical information. A set of symbols are pre-defined to represent the objects of interest. For example, a small, medium and large lake are represented by the symbols "S", "M" and "L" respectively [6]. A line segment is used as the symbol for a road [20]. A star represents a "site of interest" [26]. If there are new objects of interest, more symbols have to be specified. In contrast, the DISIMA type system provides a Polygon class to define all arbitrary shape, or objects of interest. These GIS applications handle directional relationship, but they do not handle topological relationship, which is supported by the DISIMA model. The GIS applications also lack the expressive power on color and texture.

[9] and [28] focus on the shape feature and omit the spatial, color, and texture issues. [9] discusses how to apply coded contours to indicate shape similarity, taking into account of contours within the shape boundary. The DISIMA model uses composite shapes to define such interior contours. In addition, the DISIMA model allows applications to separate regular shapes, i.e., circle, square, ellipse, etc., from arbitrary shapes, so that the query result can be more precise, and the search can be limited to the desired type of shape. [28] applies shape matching in two stages: the fast pruning stage, to reduce the initial size of the search space, and the deformable template matching stage, to perform the final match. However, it does not support composite shapes.

Most of the papers on image database models, such as those discussed above, are specific to either color, shape or texture, and do not provide a complete view. Based on the information obtained from the web-site [1], the photobook system covers more image features. Their research includes texture modeling, face recognition, shape matching, brain matching and interactive segmentation and annotation. Their work is more oriented toward image processing, but there is no one system, which integrates individual components into a framework, like the DISIMA type system. Another difference between the DISIMA model and other models is that the latter do not have a semantically rich underlying query language, which can efficiently handle image queries on color, shape, texture and spatial relationship.

Spatial relationship can be subdivided into directional and topological. While directional relationship is the focus of GIS applications, topological relationship is one of the main concerns in medical applications. For example, knowing that the tumor is on the left of the lung is not enough, a medical doctor needs to know whether the tumor and the lung are disjoint, met, overlapped, etc. A semantic data model is therefore evolved for medical applications ([8]). This model applies object contour and spatial relationship, together with the temporal element, to analyze medical images. Although [8] has the temporal feature, which has not been implemented in the DISIMA model, this medical model is too restrictive. It lacks the other image features, such as color and texture. The DISIMA model has a generic and flexible design, which can serve a wider application domain.

The one system which is more comparable with DISIMA is IBM's QBIC [15]. Both QBIC and DISIMA provide a graphical user interface (GUI) in Java, and are therefore accessible through the Internet. QBIC has a video component while the video portion of DISIMA has not yet been developed. In addition, QBIC allows textual information to be obtained by clicking an image, while this is still being developed in DISIMA. With respect to the work of this report, the following comparison is made based on the information available at the QBIC web-site [3] and from [15].

- Spatial relationship is one of the image features in DISIMA, but it is not supported by QBIC.
- DISIMA supports composite objects, while QBIC does not.
- DISIMA allows an object to associate with different shapes, representing different zoom levels, while QBIC does not.
- The similarity threshold of a query can be defined by the user in DISIMA, but QBIC does not have this option.
- An outstanding feature of DISIMA is its underlying query language, MOQL, which is semantically rich. A MOQL query can be executed by the query engine as a structured expression without using the graphical interface.
- The work of this report supports an integrated graphical interface, allowing the different image features such as spatial, color, and shape to be specified in a query. Instead of allowing the user to describe the image features, QBIC displays some sample images and limits the retrieval to images which are similar to those displayed. In this respect, DISIMA provides a more flexible approach.

Chapter 3

DISIMA Kernel

The DISIMA project aims at building an image database system which supports content-based queries combining spatial and other image features such as shape, color and texture. The research is initially on still images and will be extended to cover video and distribution. Many models in the literature are designed for specific applications. The DISIMA model aims to provide a generic content-based type system which can be customized to meet the requirements of most image database applications. DISIMA system will be distributed, enabling users to retrieve images located at different databases. The DISIMA model includes a graphical user interface available on the World Wide Web through which MOQL queries can be generated and executed. Detailed discussion of the DISIMA model can be found in [22].

3.1 Terminology

In the DISIMA type system, an image is viewed as a collection of objects, but not every object in the image is of interest to an application. Objects that are of importance to the application are called *salient objects.* Salient objects are divided into *logical salient objects* (LSOs) and *physical salient objects* (PSOs).

3.1.1 Logical Salient Object (LSO)

A *logical salient object* can exist in the database even if there is no image in the database referring to that LSO. For example, "Clinton" is a person (LSO) which has data attributes lastname, firstname, political party, etc. Clinton's LSO can exist even if there is no image of him created in the database.

3.1.2 Physical Salient Object (PSO)

LSOs and PSOs have a one-to-many relationship. Each PSO corresponds to an appearance of the LSO in an image. For example, "Clinton" may have two images in the database, one with his family and the other with "Chretien". In this case, two Clinton's PSOs are constructed in the database corresponding to one Clinton LSO.

3.2 A Content-Based Generic Type System

The goal of a content-based type system in image databases is to support data storage and query execution based on image content, i.e., shape, color, texture and spatial relationship. The main difference between a traditional entities-relational database system, and a content-based database system is that the former uses exact match while the latter uses similarity match. Exact match returns either true or false. If the answer is true the image is retrieved; otherwise it is not retrieved. In contrast, similarity match applies distance functions to compare database images with the target image, and assigns grades between 0 to 1 (1 means an exact match) to the images, in the database. The images that have grades greater than a specified threshold are retrieved and displayed to the user in descending grade order.

As explained in Section 2, current image database systems are mostly designed for specific applications. In contrast, the type system discussed in this chapter is generic and is designed to address most of the issues in image databases. In other words, it is designed to support color, shape, texture and spatial relationship, in addition to the traditional keywords. The temporal feature will be included at a later stage. Futhermore, the type system can be customized so that an application can decide to implement one or more of the image features. Applications can also expand the LSO hierarchy to add user-defined classes, such as Person, Building, Animal, and so on.

3.2.1 The Type System Overview

Figure 3.1 shows a high level view of the classes used in the DISIMA type system. The Mbb (Minimum bounding box) class defines the spatial feature and the Geometric_Object class defines the shape feature.

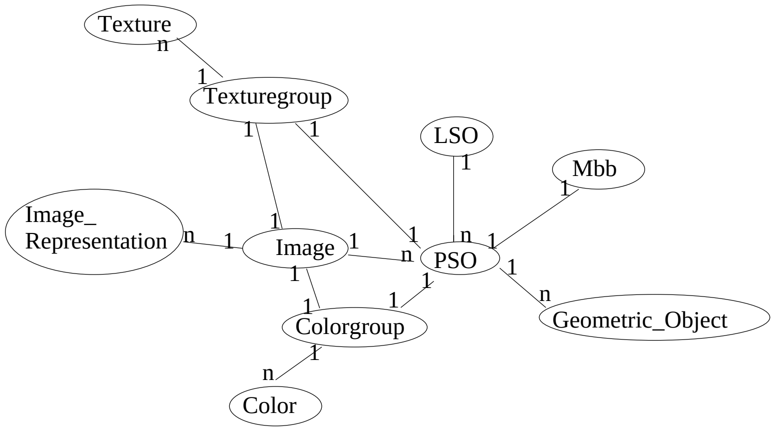


Figure 3.1: Content-Based Generic Type System

In relational databases, query execution requires the joining of tables. The final result is obtained by selecting the right tuples and projecting the right attributes. In object Database Management Systems (DBMSs), instead of the join-operations, *the associative* and *navigational* techniques are applied based on oids. In the DISIMA type system, the PSO class acts as a bridge to access other classes. Each PSO instance contains oids which are linked to the corresponding LSO, Mbb, Geometric_Object, Texturegroup, Colorgroup and Image instances.

Another advantage of using the *star* shape design, which places PSO at the center of the type system, is that applications can choose not to implement some of the image features, without losing the integrity of the remaining features. For example, a GIS application may omit the color and texture features, while an application designed for fabric manufacturers may omit the spatial feature.

The design of the type system can be customized for an application, if required, by associating the Colorgroup class with the Geometric_Object class so that each shape, instead of each PSO, can have its own colorgroup. The disadvantage of this approach is that applications will lose the flexibility of implementing the color feature without implementing the shape feature.

The DISIMA type system allows a PSO to associate with more than one shape (shown by the 1-to-n relationship). This is necessary if an application allows the shape of an object to vary in a way similar to how human eyes perceive objects at different zoom levels. A simple example is that a circle will become a point when viewed at a distance. In this case, both the point and circle are stored as shapes of the PSO.

3.2.2 Logical Salient Object (LSO) and Physical Salient Object (PSO)

The concept of LSOs and PSOs will become clear by examining figure 3.2. There are other objects in the background but suppose only "John Kennedy" and "Marilyn Monroe" are of interest to the application and thus there are two LSOs constructed in the database. Every appearance of "Kennedy" in an image corresponds to a PSO and so does every appearance of "Monroe". Therefore there are three PSOs in total, with one associated with "Kennedy" and two associated with "Monroe".

Figure 3.2: Each LSO can be associated with more than one PSO

An LSO instance maintains the textual information, such as Kennedy's lastname, birth-year and political partyname. The LSO can exist independent of the PSOs, but before creating any PSO, the LSO has to be in the database.

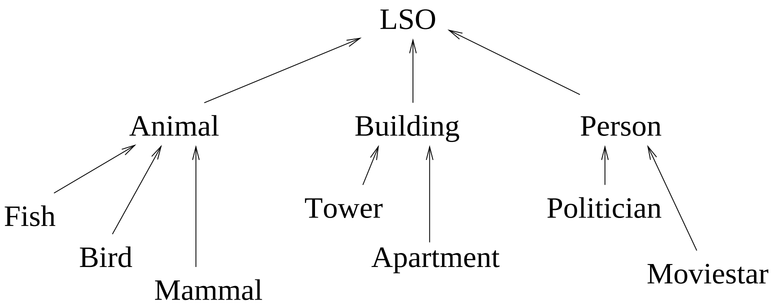


Figure 3.3: An example of the LSO class hierarchy

To make use of inheritance in OO design, the LSO classes are defined in a hierarchy with superclasses and subclasses so that a subclass can inherit attributes and methods from its superclass, as shown in figure 3.3.

3.2.3 Spatial Relationship

Since an image is viewed as a collection of objects, when more than one object appears in an image, spatial relationships are established. Spatial relationship refers to the distance, directional relation and topological relation between objects [18]. Distance is measured between the centroids of objects. The centroid of an object is the point (x_{avg}, y_{avg}) such that,

$$x_{avg} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{3.1}$$

and

$$y_{avg} = \frac{\sum_{i=1}^{n} y_i}{n} \tag{3.2}$$

where n is the number of pixels (points) forming the object.

Topological relation can be estimated based on the Mbbs of objects, or computed using the actual regions of the objects. Directions can either be measured in degrees from 0 to 360, or by using the eight directions (north, south, etc.) The DISIMA type system applies the Mbb approach, and the definitions on directions and topology defined in the MOQL extension [18], to measure spatial relations.

The Minimum Bounding Box (Mbb)

The Mbb approach has been studied and used in research projects to estimate the spatial relations between objects. The Mbb approach is popular because it is computationally simple and storage efficient. In order to construct the Mbb of an object, the best fit rectangle horizontal to the x-axis is drawn to enclose the object. When putting an image in the context of an X-Y plane, the top-left corner of the image is used as the original. The Mbb is defined by taking the upper left (X_{min}, Y_{min}) and lower right (X_{max}, Y_{max}) corners of the rectangle.

Topological relation

There are eight types of topological relations: *disjoints, meets, equals, overlaps, contains* (converse is *inside*) and *covers* (converse is *covered_by*).

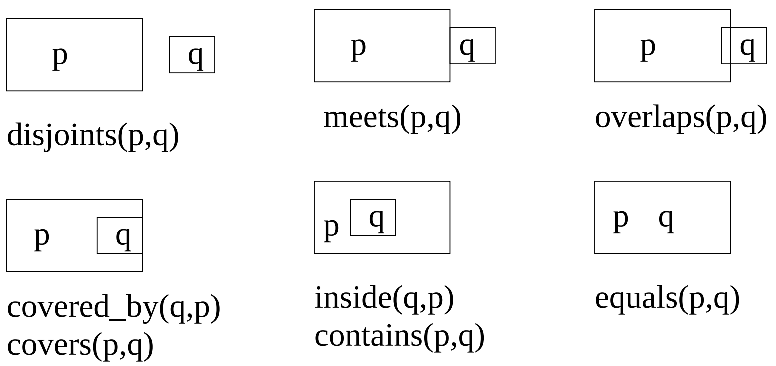


Figure 3.4: Topological relations

The topological relation between two objects is estimated by comparing the X_{max}, X_{min}, Y_{max} and Y_{min} of the two Mbbs.

It is important to understand that the Mbb approach works well only for shapes evenly distributed in the Mbbs like those in Figure 3.5 (left), where the mbbs show that the objects (head and hand) overlap. However, for shapes not evenly distributed, an estimation based on Mbbs can give a wrong result. Figure 3.5 (right) shows that, although the photo and the pen are disjoint, their Mbbs meet.

For this reason, the topological relation estimated by the Mbbs does not necessarily imply the true relation of the objects. In order to obtain an accurate result, more than one topological relation has to be considered. For example, to identify objects which are disjoint, not only disjoint Mbbs but also Mbbs which overlap, meet, cover and contain should be retrieved and analyzed further. Two disjoint objects, with Mbbs showing different topologies, are illustrated in Figure 3.6.

In spite of this discrepancy, the Mbb approach is still useful as a preliminary filter to eliminate most of the unqualified objects during query execution. The resulting objects can be further analyzed by more accurate but computationally complex algorithms. Without the preliminary filter, query execution will

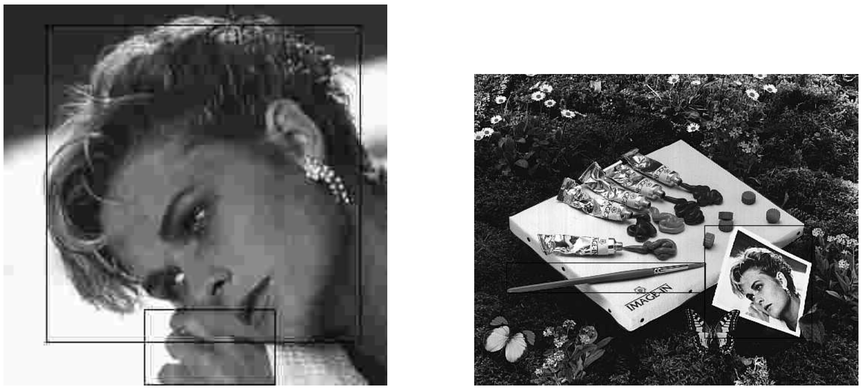


Figure 3.5: Mbbs may not reflect true topology

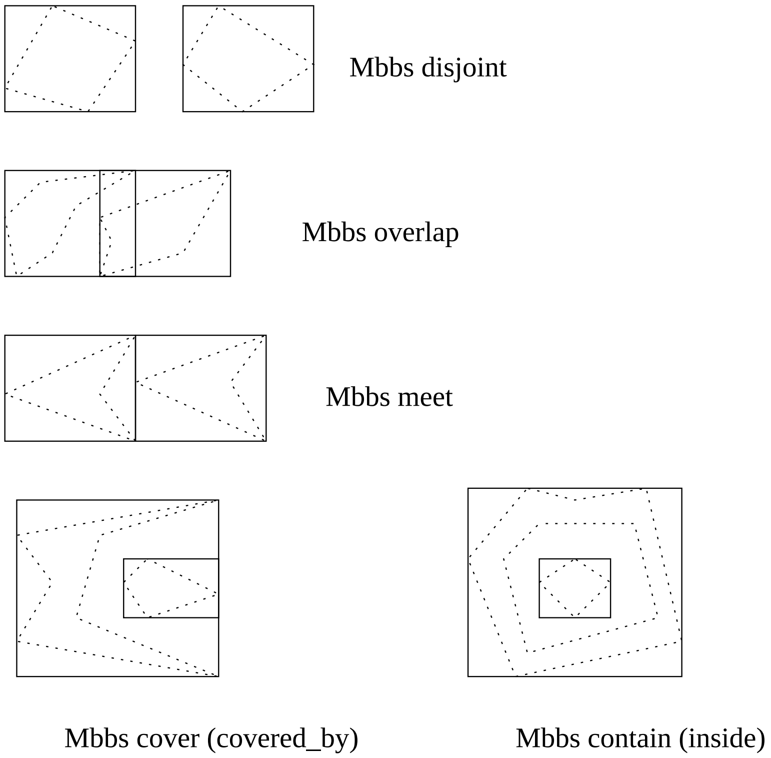


Figure 3.6: Inconsistency between Mbbs and objects topology

take longer due to additional computation.

Directional relation

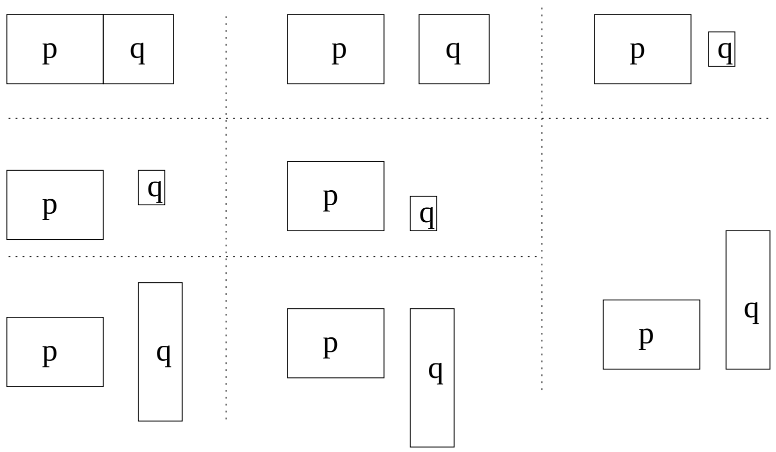


Figure 3.7: An example of using Mbbs to measure direction, i.e., q is east of p

The directional relation is valid only when two objects are disjoint or meet [18]. If two objects overlap, the spatial relation is defined by their topology. Figure 3.7 shows how the Mbb approach is used to measure direction.

The relative direction of one object from another can be defined as *north, south, east, west, northeast, northwest, southeast and southwest.* In a 2-dimensional image, above, below, left and right are generally interpreted as north, south, west and east. Three dimensional spatial relationship is not yet defined in the DISIMA type system.

Methods to support the Mbb approach: the Interval class

In order to derive the topological or directional relation between two Mbbs, the x and y coordinates, or intervals, of the Mbbs are examined separately. The Interval class in the DISIMA type system provides methods to compare intervals. These methods are *before(), equal(), during(), start(), finish(), meet()* and *overlap().*

Each of these functions returns true or false when comparing two intervals. The topological and directional relations are established based on the true values of a combination of these functions. For example, given two objects A and B, Mbb_A is above Mbb_B if:

$A_{Y-interval}$ is *before* $B_{Y-interval}$ or
$A_{Y-interval}$ *meets* $B_{Y-interval}$.

Defining Mbbs in terms of Intervals
The definitions of spatial relationships can be found in [18], where comparison of Mbbs is divided into two categories:

- Directional relations include *left, right, above, below, front, back, south, north, west, east, northwest, northeast, southwest, southeast,* as well as the combination of *front* and *back* with other directional relations, i.e., *front_left.*

- Topological relations include *inside, cover, touch, overlap, disjoint, equal, coveredBy,* and *contain. Cover* and *coveredBy* are inverse of each other and same as *inside* and *contain.*

An Mbb is defined by its upper-left and lower-right corners. It is always drawn horizontal to the x-axis and y-axis. By examining the x and y coordinates separately, an Mbb can be represented by its x-interval and y-interval. Spatial relationships between two Mbbs can be determined by comparing their x and y intervals, respectively.

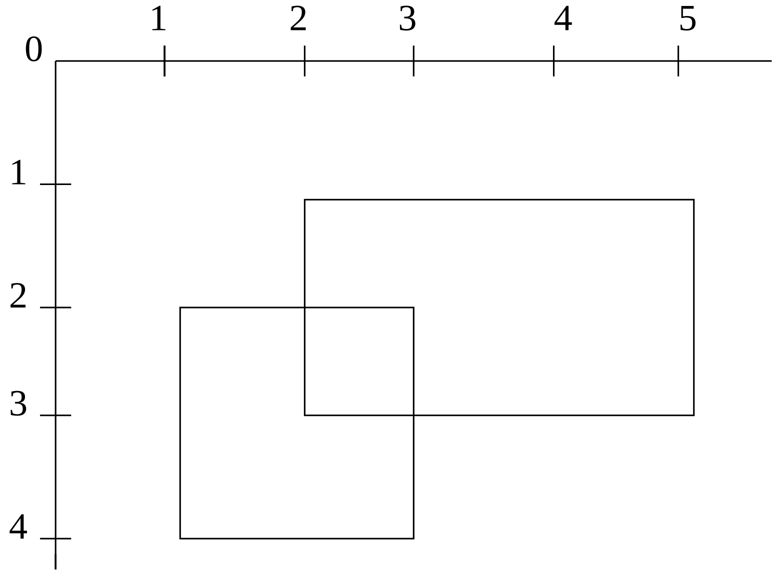


Figure 3.8: Mbb defined by intervals

Figure 3.8 follows the convention of an image with the original at the top-left corner. One of the Mbbs has x-interval [1,3] and y-interval [2,4]. The other Mbb has x-interval [2,5] and y-interval [1,3]. The spatial relationship between the two Mbbs is described as *overlaps* because both the x and y intervals of the two Mbbs overlap each other. If only the x or y interval overlaps but not both, the spatial relationship is described as *disjoint* or *north, south, west, east, above, below,* etc. depending on the relative positions of the intervals. The primitives used to compare two x-intervals are illustrated in Figure 3.9. Based on the x and y interval primitives, the directional and topological relations between two objects can be determined.

MOQL also applies the interval primitives to model temporal elements, i.e., time. For example, if the time interval of video-clip1 is before the time interval of video-clip2, then video-clip1 is *before* video-clip2.

Implementation Considerations

In order to extract image features and store the data in the database, sophisticated image processing techniques are necessary. One of these techniques is *edge detection,* which outlines the shape of an object. From the image processing point of view, each object is a collection of pixels. Depending on the gray-level or color of the pixels, not all of them can be detected accurately. Figure 3.10 shows that the Mbb may not be able to enclose the object because some pixels at the boundary are not detected accurately. This will affect the Mbb approach because the Mbbs may be disjoint while the objects overlap. To allow flexibility in the system, the *limit* variable can be defined by the application to set the tolerance limit when comparing intervals. For example, to decide whether two x-intervals x_A and x_B are equal,

$$| x_A - x_B | \leq limit \tag{3.3}$$

can be examined instead of

$$| x_A - x_B | = 0 \tag{3.4}$$

where $limit$ can be ≥ 0.

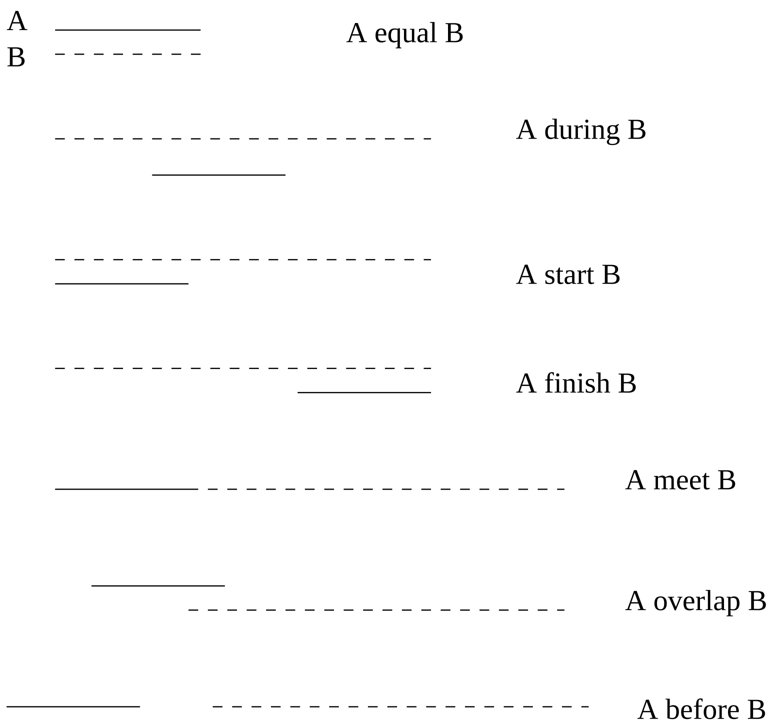


Figure 3.9: How x-intervals are compared

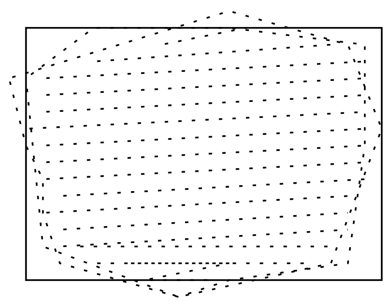


Figure 3.10: Edge detection may not be accurate

3.2.4 The Color Feature

The color feature can be associated with an object (e.g., apple), or an image (e.g., the sunset scene).

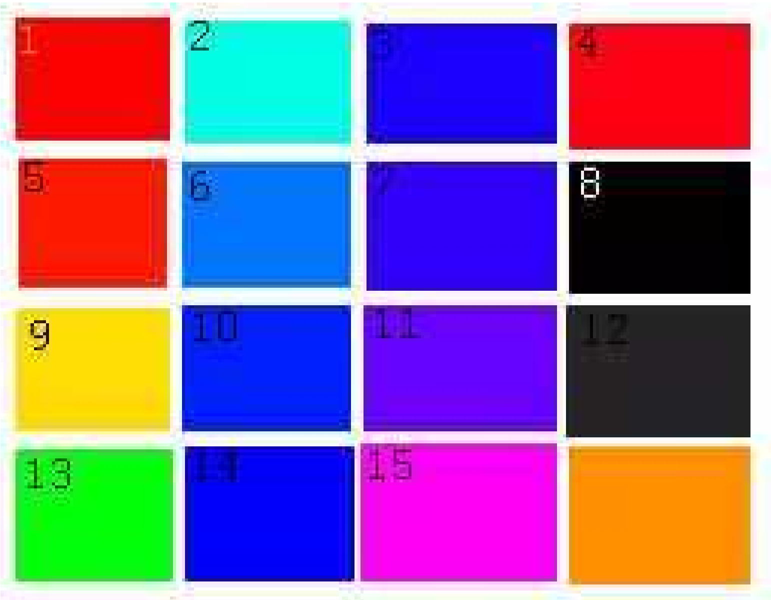


Figure 3.11: The red, green, blue components of a color

Each color is defined by its three component values: red (R), green (G) and blue (B). The RGB components of the colors in Figure 3.11 are analyzed in the following table.

Color	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Red	255	0	25	255	255	0	47	0	255	0	103	36	0	0	255	255
Green	0	255	0	0	25	117	0	0	220	31	0	36	255	0	0	142
Blue	0	229	255	22	0	255	255	0	0	255	255	36	8	255	247	0

There are two common ways to store the color feature. The first method is to associate a color identity (id) with an object, and use the color id (key) to reference the corresponding color values in a lookup table. The second method is to store the red (R), green (G) and blue (B) values of a color directly with an object. The RGB values are in the range [0,255] and therefore can be stored as a CHAR structure.

The first method requires less storage space because each color is stored only once in the lookup table. On the other hand, the second method is more time efficient because there is no need to retrieve the color values from the lookup table during query execution. In order to shorten response time, the DISIMA type system has chosen the second method.

To allow an object or an image to associate with more than one color, the color feature is defined by the Colorgroup class in the type system. Each colorgroup instance is made up of one or more color instances.

Depending on the requirements of an application, different color extraction techniques can be applied. The simplest algorithm is to take the average color (avg_R, avg_G, avg_B) of a region.

$$avg_R = \frac{\sum_{i=1}^{n} r_i}{n}, \tag{3.5}$$

$$avg_G = \frac{\sum_{i=1}^{n} g_i}{n} and \tag{3.6}$$

$$avg_B = \frac{\sum_{i=1}^{n} b_i}{n} \tag{3.7}$$

where n is the number of pixels in the region and (r_i, g_i, b_i) are the color values of the i^{th} pixel.

Another technique is to extract a few dominant colors to represent a region. Since the type system can store a colorgroup, this technique is possible. Currently, only one color (the average color) is stored with each object. Later in the development, the average color will be replaced by a number of dominant colors.

3.2.5 Geometric Object Hierarchy (The Shape Feature)

If all the objects can be described using the primitive shapes such as circle, square, rectangle, triangle, etc., processing will be easier. Unfortunately, the type system has to take into account arbitrary shapes which cannot be described in simple geometry. To describe arbitrary shapes, the type system uses polygon. Imagine repeatedly cutting the boundary of a shape into half. It will come to a stage where the divided portions are very close to straight lines (Figure 3.12). The polygon formed by these segments can give a reasonably accurate estimation of the shape. The representation becomes more precise when the number of vertices of the polygon increases.

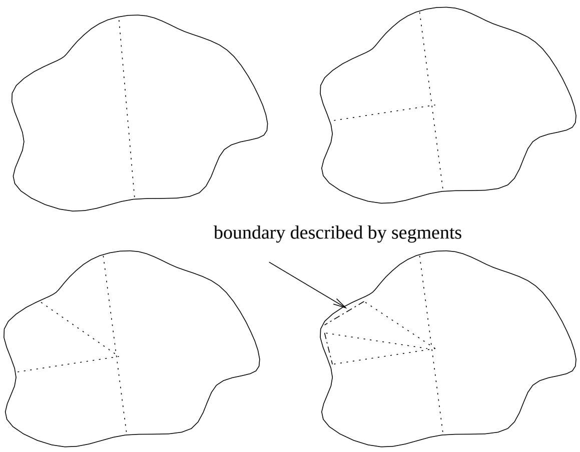


Figure 3.12: Uses polygon to describe arbitrary shape

Why does the type system define other primitives if shapes can be represented by polygons? This is because some applications, i.e., graphic design, may need precise descriptions of circle, ellipse and other basic geometric shapes. Separating these primitives from the general Polygon class also reduces the search and computation time because the algorithm applied to shape matching can be computationally complex (refer Section 3.3).

Atomic Shape

A geometric shape can either be atomic or composite. Atomic shapes are further categorized into: point, 1-dimensional and 2-dimensional. 1-dimensional shapes do not have area. 2-dimensional shapes have area but not volume. 3-dimensional shapes can be incorporated into the type system in the future.

Definition of Atomic Shapes

When defining atomic shapes, the image is viewed as an X-Y plane with the origin at the top-left corner. X increases from left-to-right and Y increases from top-to-bottom. Measurement is made in terms of pixels.

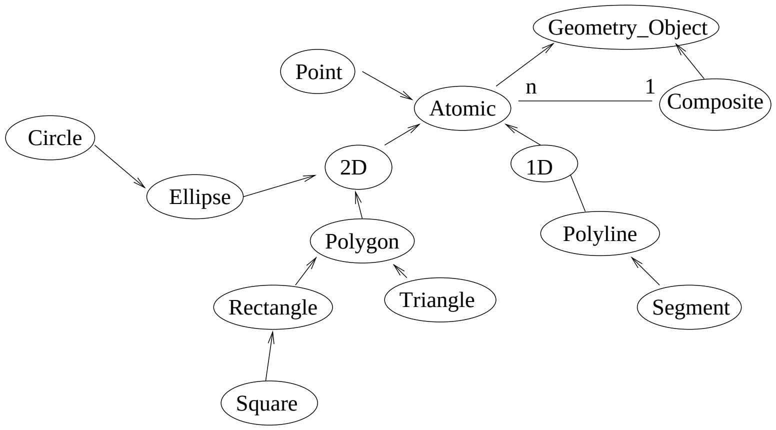


Figure 3.13: The Geometric Object Hierarchy (Logical design)

- A **point** is defined by its x and y coordinates on the X-Y plane.
- A **polyline** contains n consecutive straight lines with no intersection, i.e., $n \in [2,\infty]$, and the start point is not equal to the end point.
- A **segment** is a polyline with $n = 2$.
- An **ellipse** is defined by its center and the end points of its major and minor axes on the boundary.
- A **circle** is an ellipse with the two axes of equal length.
- A **polygon** contains n consective segments, i.e., $n \in [3,\infty]$, and the start point is equal to the end point.
- A **triangle** is a polygon with $n = 3$.
- A **rectangle** is a polygon with $n = 4$, and each segment is 90^o clockwise from the previous one.
- A **square** is a rectangle, but the four segments are of the same length.

The Atomic class can be extended to incorporate more shapes, such as curve or arc, if required.

Design and Implementation Conflicts

In the OO approach, a subclass is more specific than its superclass. In other words, the subclass is defined by adding more data members or functions. From the data member point of view, Ellipse should be a subclass of Circle because a circle can be defined by its center and a point on the boundary, while an ellipse has to be defined by three points: the center and the two end points on its axes. However, from the logical point of view (Figure 3.13), Circle should be the subclass of Ellipse because a circle is defined by imposing an additional restriction on an ellipse; that is, the two axes have to be the same length. A similar argument applies to the Rectangle and Square classes.

To overcome this conflict, a flat hierarchy was once considered (Figure 3.14). The problem with the flat hierarchy is that it does not take full advantage of the OO inheritance property. After consultation with some OO design experts, the final design as shown in figure 3.15 evolved.

Figure 3.15 shows the final design of the Geometric_Object class hierarchy. The I_ class defines the data members, and the C_ class is a concrete class that defines the constructor, and other function members necessary for the manipulation of the objects. The C_ class inherits the data members from the

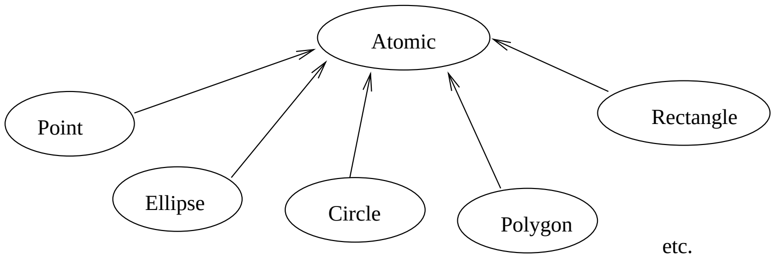


Figure 3.14: The Atomic Hierarchy (Flat hierarchy)

I_ class, and a set of virtual functions from its abstract superclass. For example, C_Ellipse inherits the data members from I_Ellipse, and the methods from its abstract superclass Ellipse. The separation of data members and methods, in I_ and C_ classes, preserves the logical concept of Circle being a subclass of Ellipse and, at the same time, there is no conflict in data members' and methods' inheritance. An extraction of the I_ class and C_ class implementation is included in Appendix A. In the current design, the I_ classes are independent from each other. An alternative design is to apply inheritance on the I_ classes, so that I_Ellipse is a subclass of I_Circle, I_Square is a subclass of I_Rectangle, and so on.

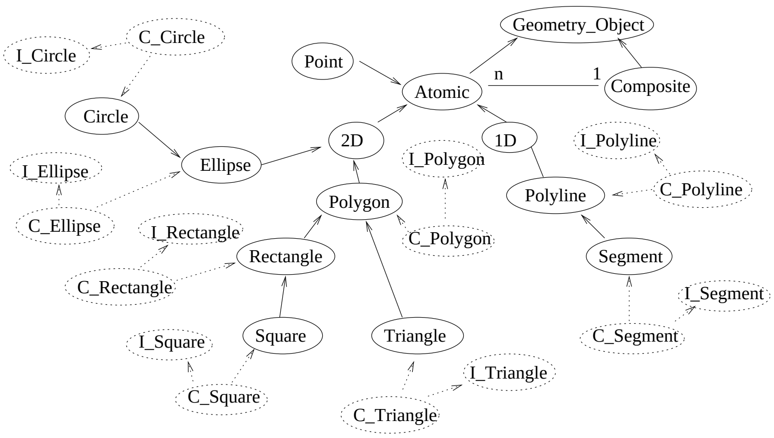


Figure 3.15: The Geometric Object Hierarchy (Final design)

A shape (Geometry_Object) can be composite or atomic. A composite shape is comprised of more than one atomic shape. Atomic shapes are further divided into three categories: point, 2-dimensional (2D) and 1-dimensional (1D) shapes. A shape which has an area, e.g. a polygon, is classified under 2D. A 1D shape has length but not area. Due to the special characteristics of the point shape, it is not classified under 2D or 1D. In the current design, two class groups are defined under 2D: the *Polygon* and *Ellipse* groups. The *Polyline* group is defined under 1D.

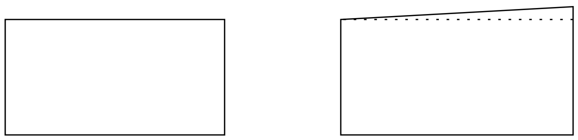
The Geometric_Object class supports three types of similarity match: *full-group, class* and *sub-group,* depending on the similarity threshold specified in the query.

- Full-group match–
  When the similarity threshold specified < 1, the query engine searches all the classes in the group. The *ellipse* group includes the Ellipse and Circle classes, and the *polyline* group includes the Polyline and Segment classes. The Polygon, Rectangle, Square and Triangle classes belong to the *polygon* group. For example, the query

```
SELECT image
FROM image m, lso o
```

```
WHERE m contains o
AND o.shape similar rectangle similarity 0.5;
```

will retrieve not only the shape on the left in Figure 3.16, but also the one on the right. The shape on the right is a polygon, but is similar to the target rectangle, and thus should be returned to the user. It is retrieved by performing a full-group match. A class match, on the Rectangle class only, will not retrieve this shape.



rectangle polygon

Figure 3.16: An example of deep search

Full-group match applies when the similarity threshold in the search condition < 1. Class match or sub-group match applies when the similarity threshold $= 1$ (exact match), depending on whether coordinates of the shape are specified.

- Class match–
  Class match searches only the class specified. For example, the condition:
  ```
  o.shape similar rectangle(1,2 10,1 10,3) similarity 1.0;
  ```
  requests a rectangle defined by the given coordinates. Since the simialrity required is 1, there is no need to search classes other than the Rectangle class.

- Sub-group match–
  However, an exact match can be expressed in a different way, without giving the coordinates. For example, the condition:
  ```
  o.shape similar rectangle similarity 1.0;
  ```
  requests any shape which qualifies as a rectangle. A class match is insufficient in this case, because Square is a subclass of Rectangle, and squares should also be returned to the user. A sub-group match searches not only the class specified, but also the subclasses.

Orientation of Atomic Objects

The translation, rotation and scaling of objects should not affect the result of similarity match. However, to allow for future requirements, the DISIMA type system is designed to store orientation information without affecting the efficiency of the system. For example, instead of storing the width and length (or the top-left and bottom-right corners) of a rectangle, in the type system a rectangle is defined by its center and two neighbouring corners. Similarly, an ellipse is defined by its center and the two intersecting points, that its major and minor axes meet the boundary. If only the width and length are stored, the rectangle is assumed to be parallel to the x-axis and the orientation information is lost. For example, Figure 3.17 shows $rectangle_A$ and $rectangle_B$, with different orientation. The orientation is defined by the angle between the principal axis and the x-axis. The principal axis can be calculated based on the center and the two neighbouring corners of a rectangle. In this example, $rectangle_A$ has an orientation angle $\theta_A \approx 45^o$, and $rectangle_B$ has an orientation angle $\theta_B = 90^o$. If orientation is specified in a query, similarity can be determined by comparing θ_A and θ_B.

If only the width and length of a rectangle is stored, $rectangle_A$ is the same as $rectangle_B$ because they have the same width and length. Some systems use the top-left and bottom-right corners to define a rectangle. This is obtained by rotating the rectangle about its center, so that the rotated rectangle is

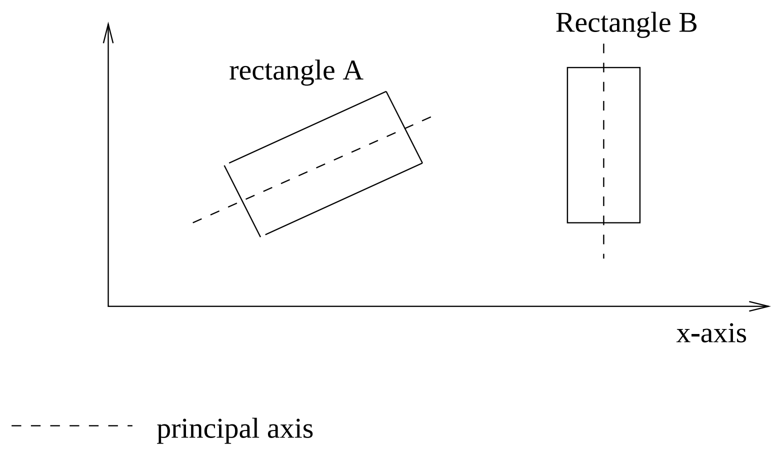


Figure 3.17: Orientation of shapes

parallel to the x-axis. The top-left and bottom-right corners are then recorded. The orientation of the rectangle is lost by the rotation–$rectangle_A$ also becomes the same as $rectangle_B$.

Composite Shape

A composite object is comprised of more than one atomic shape. There are two reasons why composite shapes are included in the type system. First, the edges of an atomic shape do not intersect and thus the Atomic class is insufficient to define shapes similar to those illustrated in Figure 3.18 (a) and (b). The *composite* shape in (a) is composed of two triangles and in (b), the shape is composed of a triangle and a polyline.

The second reason that composite shapes are included in the type system is because an application may want to group disjoint atomic shapes together to represent an object. Figure 3.18 (c), which contains three circles, one triangle and an ellipse, is of this kind.

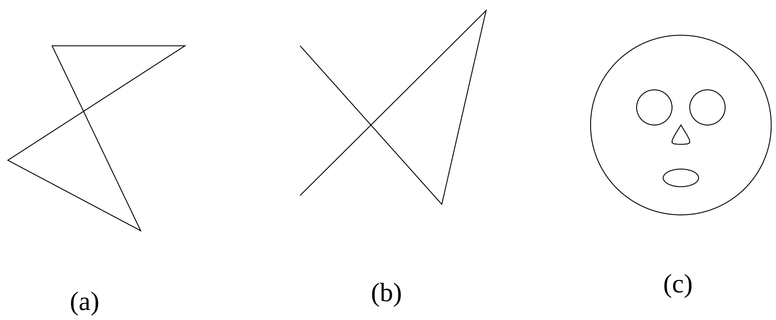


Figure 3.18: Examples of composite shapes

Methods to support Shape Manipulation

In the DISIMA type system, the translating, scaling or rotating of an object is defined as shape manipulation. There are two reasons for shape maniupulation. First, the user may want to do a what-if operation by manipulating the shape of an object in an image, and view the result on the graphical interface. Either the original shape is restored or the modified version is saved in the database after manipulation. The second reason for shape manipulation is similarity match. The target shape may need to be manipulated in order to match the database shape. In order to support shape manipulation, the type system defines the *Vector* class.

As discussed earlier, shapes can be inscribed by polygons. A polygon is defined by a list of segments or a list of points. By manipulating these segments or points, the polygon can be translated, scaled or rotated. For example, given a point (5,5) and a vector $\vec{\begin{pmatrix} 2 \\ -3 \end{pmatrix}}$, the point is translated to (7,2).

<u>Methods of the Vector class</u>
The following vector operations are designed to support the manipulation of points and segments.

- magnitude() computes the magnitude of the vector.
- gradient() computes the gradient of the vector.
- reverse() reverses the direction of the vector.
- add() computes the sum of two vectors.
- subtract() computes the difference of two vectors.
- dot_product() computes the dot_product of two vectors.
- angle() computes the acute angle between two vectors.
- clockwise() determines whether the next vector is turning clockwise from the previous vector.

<u>Methods of the Point class</u> The following methods support the manipulation of points.

- distance() computes the distance between two points.
- get_vector() computes the vector from the origin to a given point.
- rotate() computes the rotated point, given a pivot and an angle or rotation.
- reflect() computes the reflected point, given a pivot of reflection.
- scale() computes the scaled point, given a pivot and the scaling percentage in both the x and y direction.
- translate() computes the translated point, given the vector of translation.

<u>Methods of Segment class</u> The following methods support the manipulation of segments.

- get_intersect() computes the intersection point of two segments.
- parallel() determines whether two given segments are parallel.
- perpendicular() determines whether two given segments are perpendicular.
- vertical() determines whether two given segments are vertical.
- horizontal() determines whether two given segments are horizontal.
- length() computes the length of a segment.

- distance() computes the perpendicular distance of a point from a segment.
- orthogonal() computes the cutting point if a perpendicular line is drawn from a point to a given segment.
- pass_thr() determines whether a segment passes through a given point.
- intersect() determines whether two segments intersect.
- angle() computes the acute angle between two segments.

3.2.6 The Texture Feature

As with the color feature, the texture feature can be associated with an image or an object. The texture feature is defined by the Texturegroup class, and each texturegroup instance is associated with one or more texture instance. In contrast to the color instance, which defines three components: red, green and blue, a texture instance defines only one value in the normalized range [0,1]. The value is computed using an image texture extraction technique. More complicated textures require values derived from a number of dimensions. For example, to compare two floral patterns, an application may need to measure the smoothness, the horizontal, the vertical and the diagonal dimensions. In this case the texturegroup instance is comprised of four texture instances, each of which corresponds to one dimension. Other examples of complex textures are discussed in [19] which uses the dimensions: periodicity, directionality and randomness. Applications using simple texture feature may find it sufficient to use one or two dimensions. Normally, the number of dimensions improves the result but the trade-off is between accuracy and computation time.

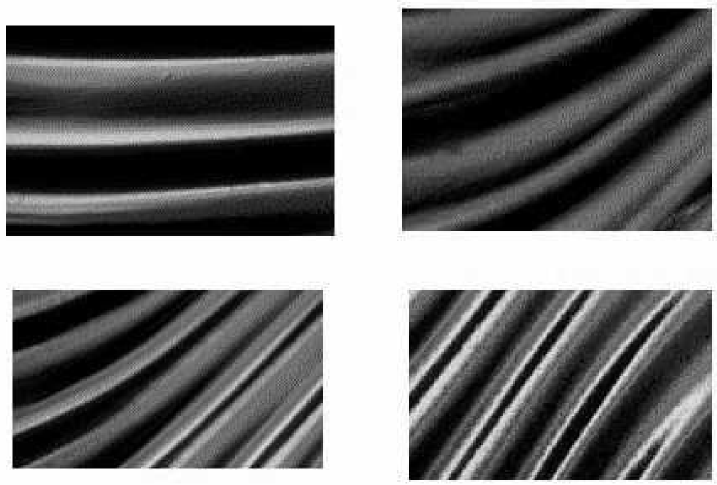


Figure 3.19: An example of texture matching

Figure 3.19 shows four images of similar texture. Suppose the top-left image is given in a query, the purpose of similarity match is to retrieve the other three from the database. The DISIMA type system is designed to store a list of texture values which can be used in similarity match (Section 3.3).

3.2.7 Image and Image Representation

The separation between Image and Image Representation classes allows different formats or compression levels of an image to be stored. An image can be stored in JPEG, GIF, BMP and TIF formats. Different file formats exist because each has characteristics that work best for a specific purpose. For example, the JPEG format works well for Internet photos, and the TIFF format works well for desktop publishing. Applications which can afford the conversion time can choose to store one basic format, i.e., JPEG, and convert to the others whenever required.

The DISIMA type system also allows the same file format to be represented at different compression levels so that the user can browse thumbnails at low resolution and display the high resolution versions only if necessary. The latter can be time consuming.

In contrast to other systems which store images in the file system, the DISIMA type system stores images in the database. The raw image is stored as a byte stream in the image_representation instance. The advantage of such design is to maintain the integrity of the database, making full use of the concurrency control, transaction processing and recovery supported by the underlying Database Management System (DBMS).

3.2.8 Methods to support Query Execution

The DISIMA type system provides a number of *query functions* to support query processing. There are three categories of query functions, as described below.

<u>QueryDeep() (Deep search)</u>
QueryDeep() is used to execute a full-group match, or a sub-group match (refer Section 3.2.5), where the query engine searches a hierarchy of classes. For example, the query

```
SELECT m
FROM image m, person p
WHERE m contains p;
```

retrieves not only persons, but also politicians, moviestars, etc., who belong to the subclasses of Person. Another example is:

```
SELECT m
FROM image m, person p
WHERE m contains p
AND p.yearOfBirth<1980;
```

which retrieves all persons, politicians, moviestars, etc., whose yearOfBirth $<$ 1980.

<u>Query() (shallow search)</u>
Query() is used to execute a class match (refer Section 3.2.5). The query engine does not search any superclass or subclass; only the class specified in the query is included in a shallow search.

<u>Equal()</u>
The similarity match algorithms are implemented in the *equal()* functions. These functions are provided by the DISIMA type system to compare the database images with the target image, and return similarity grades, which are then compared with the threshold specified in the query. If the grade is lower than the threshold, the database image is discarded; otherwise the image will be retrieved.

A grade of '1' is assigned to any successful spatial matching (e.g. p.mbb above q.mbb) or keyword matching (e.g. p.yearOfBirth=1975). The grade for shape, color or texture is computed according to the distance functions which is described in Section 3.3.

3.3 Similarity Match

Understanding similarity match requires a knowledge of image processing. Similarity match techniques are often complex and diverse. They are designed for specific applications and there is no single general algorithm which can satisfy every requirement. The purpose of this section is not to analyze all the similarity match algorithms available in the literature, but to introduce the concept and provide background information of how the DISIMA type system is designed to implement this concept. Readers interested in the topic can refer to other image processing research papers (e.g., [9], [19], [20], [27]) for detailed discussion on similarity match algorithms.

The difference between similarity match and exact match is that the former does not require the user to remember the precise detail of the image or object. In fact, certain features, i.e., shapes, are difficult to describe precisely. For example, Figure 3.20 (c) is similar to a triangle but it is not a triangle. When a query asks for a triangle, only (a) and (b) will be retrieved in an exact match operation. However, a similarity match will retrieve (c) as well.

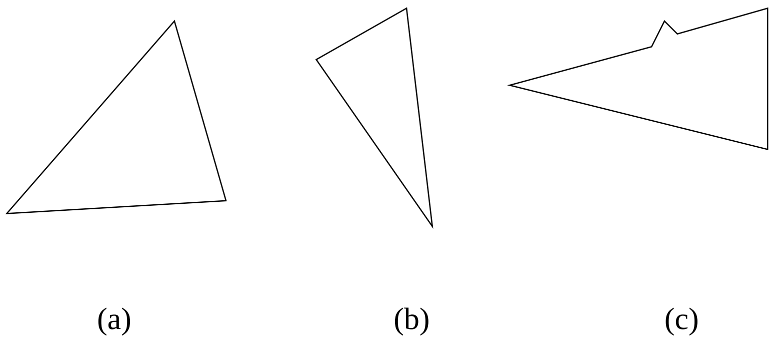


Figure 3.20: Shapes equal or similar to a triangle

Similarity match algorithms define *distance functions* which, depending on the degree of similarity, assign grades to the database objects. '0' means no match and '1' means an exact match. The user can limit the output by specifying a threshold in the query so that objects with lower similarity grades will not be extracted. The user can also limit the output by restricting the number of images displayed.

Similarity match is a useful process but it can be expensive. Complex and time comsuming distance functions are often required in order to produce accurate results. In the DISIMA type system, distance functions can be defined at condition level or at image level. An image is retrieved only if its similarity scores are higher than the thresholds both at the condition and image levels.

Applying similarity threshold to a condition

Consider the following query which has a *contains* condition, and a *shape* condition.

```
SELECT m
FROM image m, lso o
WHERE m contains o
AND o.shape similar polygon(1,1 10,2 5,12 3,20) similarity 0.8;
```

The similarity threshold 0.8 applies to the condition o.shape. If the return value of the distance function applied to o.shape ≥ 0.8, then the object is retrieved. If no threshold is specified, the default is 1.

Applying similarity threshold to an image

Different from the threshold applied to a condition, *global similarity* applied to the image.

```
SELECT m
FROM image m, lso o
WHERE m contains o
AND o.shape similar polygon(1,1 10,2 5,12 3,20)
AND o.color similar colorgroup(200,100,150)
global similarity 0.9;
```

Here, the similarity threshold 0.9 applies to the whole image (all the conditions). The query has three conditions:

```
m contains o,
o.shape and
o.color
```

After each condition, a similarity grade is scored. The general equation to calculate global similarity G is given by:

$$G = e_1 grade_1 + e_2 grade_2 + ... + e_n grade_n \tag{3.8}$$

where e_i is a coefficient and $\sum_{i=1}^{n} e_i = 1$. $Grade_i$ is the similarity score of $condition_i$. In this example, if $G \geq 0.9$, the image will be retrieved.

The ultimate goal is to allow applications to set the defaults of e_i at installation time and allow the defaults to be overriden at query level (refer Section 6.2 on future work). At the present, e_i is assigned $\frac{1}{n}$ where n is the number of conditions in the query.

3.3.1 Examples of Similarity Match Algorithms

Various similarity match algorithms on color, shape and texture have been discussed extensively in the literature; the choice of one is application dependent. An important aspect of all these algorithms is *normalization.* In order to compare the return values derived from different algorithms, these values are normalized in the range [0,1].

Algorithms on Shape

A shape is defined by its boundary. If the boundary can be represented by a mathematical formula, then the comparison can be simplified. Unfortunately this is generally not possible. Over the years, different algorithms have been suggested. The general focus is to make the comparison invariant of translation, scale and rotation. Some of these algorithms are discussed below.

- **Chain code–**
  Chain code was one of the early algorithms to compare shapes [11]. The eight directions are used as the measuring units. The boundary of an object is segmented and represented by these units.

Direction	N	S	E	W	NE	NW	SE	SW
Code	1	2	3	4	5	6	7	8

  In Figure 3.21 (a), the object is represented by the chain code (5,7,8,6) clockwise. Objects of similar shape should have similar chain codes. To make the comparison independent of rotation, after each comparison the first number is removed and placed at the end of the chain, and the two chains are compared again.

  Chain code provides only a rough measurement. For shapes like the one illustrated in Figure 3.21 (b), eight directions are not sufficient. More precise measurement can be obtained by extending the eight directions to sixteen. However, there are shapes requiring more than sixteen directions.

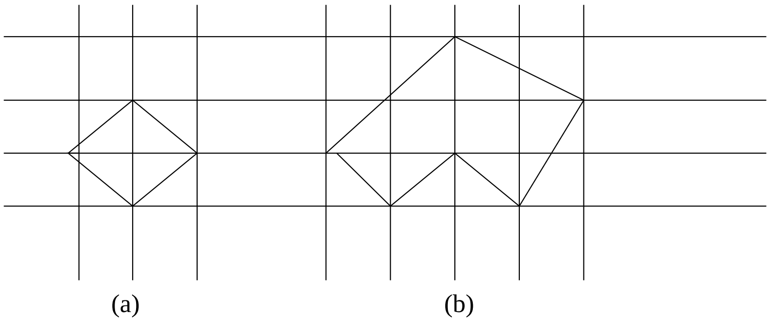


Figure 3.21: Boundary represented by chain code

- **Signatures–**
  The basic idea behind the signature algorithm is to transform the boundaries of objects into continuous functions which are easier to compare.

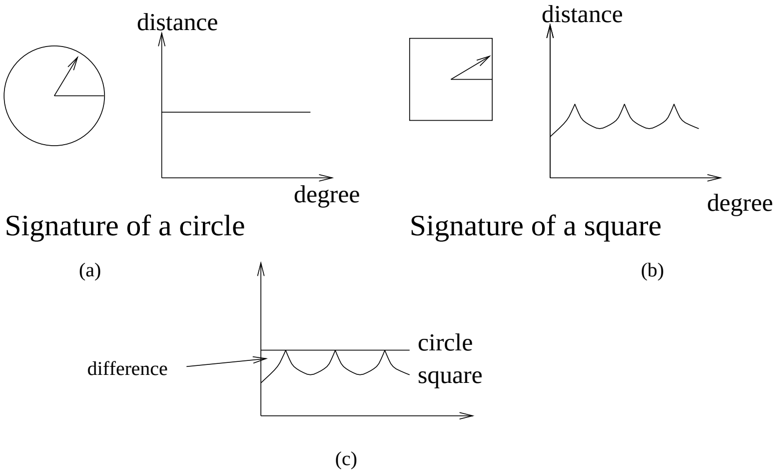


Figure 3.22: The signature algorithm

Figure 3.22 (a) shows the signature of a circle. The horizontal axis represents the measurement in degrees starting from the positive x-axis and moving counter-clockwise. The vertical axis represents the distance between the boundary and the center of the circle. When the shape is a circle, the distance (radius) is constant and thus the signature is a horizontal line from 0^o to 360^o. The signature of a square is given in (b). Figure 3.22 (c) shows the comparison between the two signatures. The similarity between a circle and a square is computed by taking the area between the signatures. The smaller the area, the more similar the shapes.

It is likely that similar shapes of different scales may appear in images, as shown in Figure 3.23. To make the comparison independent of scale, the perimeter of the shape is normalized to '1'.

The distance function is given by:

$$similarity = 1 - \frac{\sum_{\theta=1}^{n} |D(\theta)_{db} - D(\theta)_{qry}|}{n} \tag{3.9}$$

where θ is the degree anticlockwise from the x-axis and $D(\theta)$ is the distance between the boundary and the center. $D(\theta)_{db}$ and $D(\theta)_{qry}$ refer to the database object and the desired object respectively. Assuming n pair of corresponding values are selected from the two signatures for comparison, each pair will contribute $\frac{1}{n}$ to the similarity.

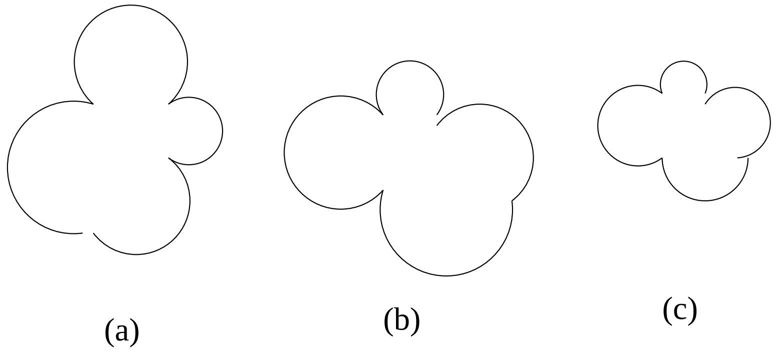


Figure 3.23: Shapes of different scales

There are two ways to make the comparison invariant to rotation. The first method is to ensure that both signatures start at the same point on the boundary, i.e., the farthest point from the center. The second method is to take many comparisons. Each comparison differs from the previous one by shifting the target signature horizontally. The minimum *similarity* computed from these comparisons is the final result.

The signature algorithm is more accurate than the chain code, but the problem is that an algorithm suitable for one type of shape may not be good for other shapes. The signature algorithm performs well to match shapes which are in proportion. When applied to distorted shapes, the similarity grade can be low even though the shapes are similar. This problem is illustrated in Figure 3.24.

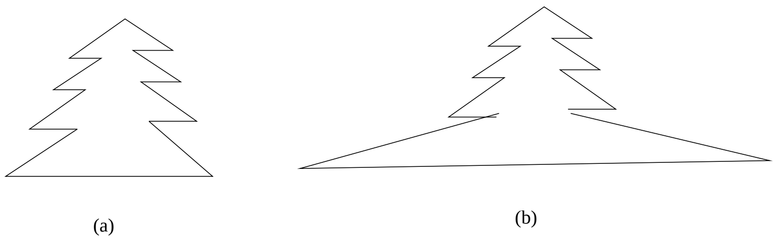


Figure 3.24: Problem with the signature algorithm

Suppose Figure 3.24 (a) and (b) are two images in the database. The upper three layers of the trees are of the same shape and in the proportion of 1:1. However, the bottom layers are out of proportion ($\approx$ 1:2). As a result of normalization, the shapes cannot match their vertices, and the corresponding values in the signatures become very different.

- **Turning angle algorithm–**
  Another algorithm discussed in the literature is the turning angle algorithm which is based on the concept that shapes can be represented by polygons.

  Take a polygon containing n edges and vertices. Starting from any point on the boundary and traversing the edge anticlockwise until a vertex is encountered. The first turning angle is the angle between the first edge and the second edge, as illustrated in Figure 3.25 (a). A total of n turning angles is recorded after traversing all the edges. Since the external angles of a polygon add up to 360^o, the sum of the turning angles should also be 360^o. As with the signature algorithm, the perimeter is normalized to '1'. By plotting the cumulative angle on the vertical axis, and the distance traversed on the horizontal axis, a step-like graph is obtained–as illustrated in Figure 3.25 (b).

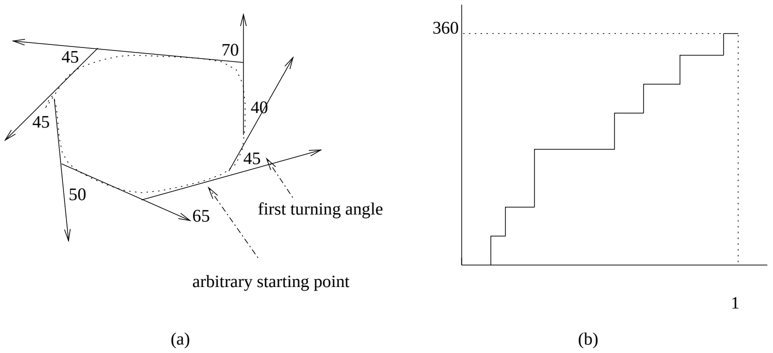


Figure 3.25: An example of turning angle

To handle shapes with concave vertices, the turning angle algorithm defines counter-clockwise as positive and clockwise as negative. In Figure 3.26, angles A, C, D and E are counter-clockwise, and angle B is clockwise. The graph contains a step down to reflect a concave angle.

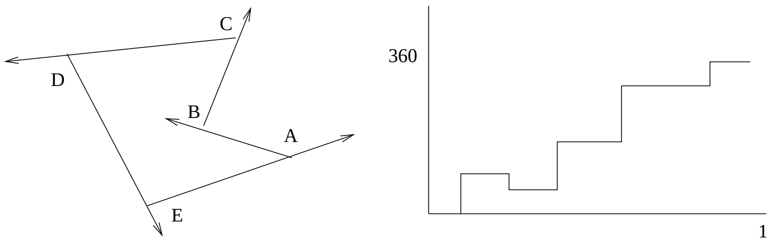


Figure 3.26: Polygon with concave angle

The advantage of this algorithm is to allow small variations in shapes. The similarity between the two triangles shown in Figure 3.27 (a) and (b) is computed by comparing the two plotted graphs. The graphs in (a) and (c) show greater difference in area reflecting a lower similarity between the two shapes. When a query asks for a triangle, the user would expect (b) but not (c) in the result.

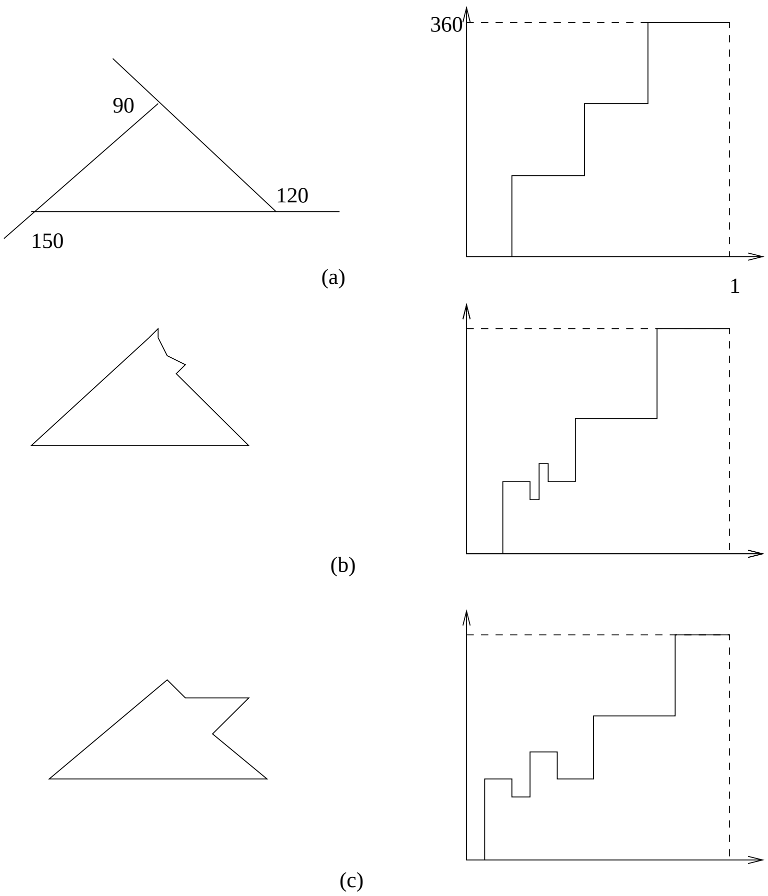


Figure 3.27: Small variation in shape

The similarity function between a database object O_{db} and a query object O_{qry} is given by:

$$min(\sum_{i=0.0}^{1.0} |\theta(db)_i - \theta(qry)_i|) \tag{3.10}$$

where $\theta(db)_i$ corresponds to the the turning angles of O_{db}, $\theta(qry)_i$ corresponds to the the turning angles of O_{qry} and min is the minumum of all the possible horizontal translations of $\theta(qry)_i$.

The turning angle algorithm is adopted by the DISIMA implementation, because this algorithm is capable of handling small variation in shapes, as illustrated in Figure 3.27.

Implemented version

The original turning angle algorithm starts from an arbitrary point on the boundary. Since in the DISIMA type system, the vertices of a polygon are known, the traversal can start from one of the vertices as shown in Figure 3.28.

By starting at a vertex V_0, the traversed distance d_i at i_{th} vertex can be simply computed by the equation $(1 \leq i \leq n)$:

$$d_i = \sum_{i=1}^{n} \frac{len_i}{L} \tag{3.11}$$

where L is the length of the boundary and len_i is the length of the i_{th} edge.

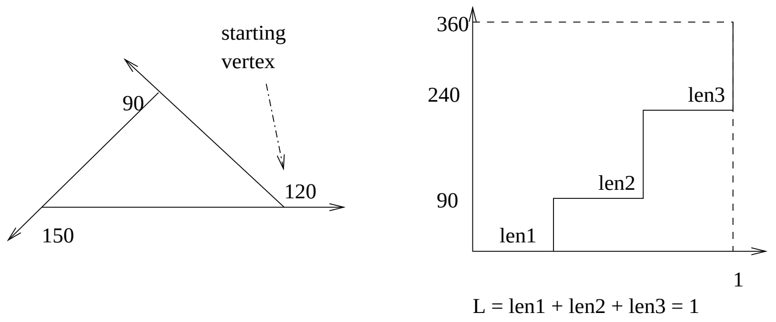


Figure 3.28: Implemented version of the turning angle algorithm

The turning angle algorithm is limited to polygons and does not apply to circle, ellipse or any curve shape implemented in the type system. If a query asks for a polygon like the one in Figure 3.29, images containing ellipses will not be included in the result.

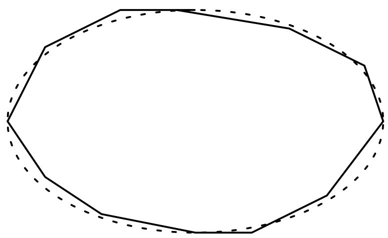


Figure 3.29: Polygon similar to an ellipse

- **Composite shape–**
  To the best of our knowledge, there is no known algorithm that compares composite shapes. The DISIMA type system proposes an algorithm which contains three stages. A higher stage produces a more precise result than the lower stage.

  Stage I compares shape
  At this 1^{st} stage, only shapes are compared, ignoring relative size and location. Suppose Figure 3.30 (a) is the target shape. The patterns shown in (b), (c) and (d) will be retrieved because they all contain a circle and three triangles.

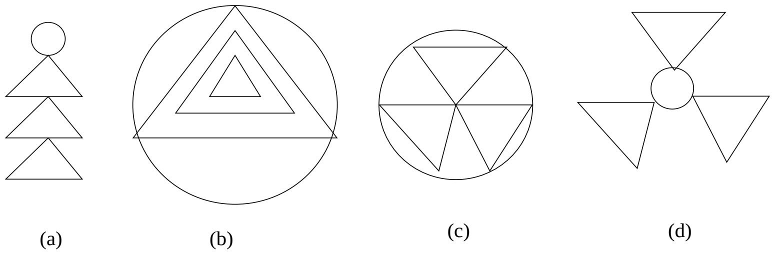


Figure 3.30: A three stages algorithm to compare composite shapes

  Stage II compares relative size
  At the 2^{nd} stage, the relative sizes of the atomic shapes are examined. Patterns (b) and (c) will be eliminated because the target shape requires the circle to be smaller than the triangles, and the triangles are of the same size.

Stage III compares spatial relations between atomic shapes

The matching of composite shapes is different from the matching of atomic shapes. The latter is independent of translation. The former has to take translation into account in order to obtain an accurate result. In stage III, the centers of the atomic shapes are used to determine their relative positions. Pattern (d) will be eliminated after this stage.

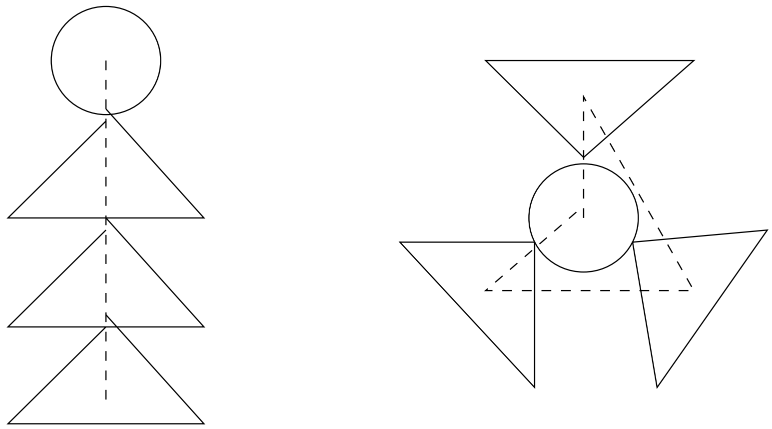


Figure 3.31: Spatial comparison on composite shapes

To perform the 3^{rd} stage comparison, the DISIMA type system makes use of the SIM algorithm ([12], [13]), which is designed for images containing multiple objects. The concept can be applied to stage III comparison of composite shapes, by treating the atomic shapes of a composite shape as multiple objects in an image. The idea is to compare the corresponding edges in the two composite shapes. The edges are formed by linking the centers of the atomic shapes, as illustrated in Figure 3.31. If there are n edges, each edge contribute $\frac{1}{n}$ to the similarity. Suppose θ is the angle between two corresponding edges, the contribution is given by the equation:

$$\frac{cos(\theta)}{n} \tag{3.12}$$

Algorithms on Color

There are two commonly used color models: the RGB and HSI models. In the RGB model, each color is defined by its red, green and blue values. Each value is in the range [0,255]. However, the RGB model is not good for color matching because it combines the pure color with intensity (or degree of brightness). The HSI model is commonly used in color matching. The HSI model separates a color into the hue, saturation and intensity components corresponding to the pure color, the amount of white light added to the pure color, and the brightness.

H is in the range [0,360], S is in the range [0,1] and I is in the range [0,255]. In order to perform similarity match, H and I are normalized to the range of [0,1]. The DISIMA type system allows applications to decide the weights to put on these components. For example, an application may emphasize the HS components and assign less or no weight to the I component. The general equation to compute the similarity between the database color (db) and the target color (qry) is:

$$similarity = 1 - (w_h h + w_s s + w_i i) \tag{3.13}$$

where

$$h = \frac{|h_{db} - h_{qry}|}{360} \tag{3.14}$$

$$s = |s_{db} - s_{qry}| \tag{3.15}$$

$$i = \frac{|i_{db} - i_{qry}|}{255} \tag{3.16}$$

The default value for the weights, i.e. w_h, w_s and w_i, is $\frac{1}{3}$. The RGB values of a region can be extracted by image processing techniques. The conversion from the RGB model to the HSI model is given by:

$$cos(H) = \frac{(\frac{((r-g)+(r-b))}{2})}{\sqrt[2]{((r-g)(r-g)+(r-b)(g-b))}} \tag{3.17}$$

$$S = 1 - \frac{3}{(r+g+b)} min(r, g, b) \tag{3.18}$$

$$I = \frac{(r+g+b)}{3} \tag{3.19}$$

where r, g and b are the red, green and blue values, respectively ([11]).

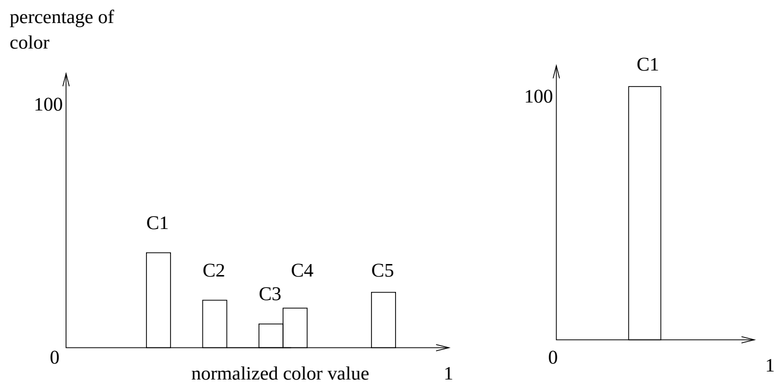


Figure 3.32: Histograms containing 5 colors and 1 color

When color matching involves a group of colors, color histograms are commonly used. Suppose the group of colors is $\{ C_1, C_2, C_3, C_4, C_5 \}$, and the percentage of C_i is p_i. Figure 3.32 (left) shows the color histogram. The similarity between two color histograms is given by:

$$similarity = \sum_{i=0}^{1} |p(db)_i - p(qry)_i| \tag{3.20}$$

where $p(db)_i$ and $p(qry)_i$ refer to the percentage of i^{th} color in the database color group and the target group respectively.

An experiment using average color

To extract all the colors in an object is a tedious task and not efficient. Instead, accurate comparison can be made based on a few dominant colors. In some cases, even average color can produce a satisfactory result. To illustrate this point, the author conducted an experiment on seventeen images, two of which were sunset images. An orange color ($r = 255, g = 142, b = 0$), which gives a sunset background, was used as the target color. Four images (Figure 3.33) were retrieved, including the two sunset images. The comparison was based on the average color of the image. There are unwanted images in the result, but the purpose of similarity match is to include the correct images. In this case, average color is sufficient for the query.

Figure 3.33: Experiment result using average color

In the current implementation, the average color inside the Mbb of an object is used. Provided the object occupies a large proportion of the Mbb, the background colors should not interfere too much with the result. This approach is also based on the assumption that there is a dominant color which is very close to the average color. This method will be replaced by using the dominant colors in the Mbb.

Algorithms on Texture

Texture is more difficult to analyze than color and shape. An example of the complex texture algorithms in the literature is the *wold model* ([1], [19]). In general, the texture of an object can be defined in different dimensions. Figure 3.19 shows a texture which can be measured horizontally, vertically or diagonally. There are other dimensions such as smoothness, coarseness, regularity, etc. A texture can be described more accurately by measuring more dimensions. However, more measurements will make computation more time consuming and inefficient. The trade-off has to be considered by individual applications.

Given a list of normalized values t_1 to t_n, taken from n dimensions, the simplest distance function is the linear average, i.e.,

$$similarity = \frac{\sum_{i=1}^{n} D_i}{n}, \tag{3.21}$$

where D_i is the absolute difference between the database t_i and the target t_i, or weighted linear sum, i.e.,

$$similarity = \sum_{i=1}^{n} w_i D_i \tag{3.22}$$

where w_i is the weight assigned to the i^{th} dimension and

$$\sum_{i=1}^{n} w_i = 1 \tag{3.23}$$

Euclidean distance can also be used, i.e.,

$$similarity = \sqrt{\sum_{i=1}^{n} w_i (D_i)^2} \tag{3.24}$$

The choice of dimension and distance function is application specific. In the current implementation, the linear average is used.

Algorithms on Spatial

Two approaches to compare spatial relationships were mentioned in Section 3.2: based on Mbb and on centroid. If the centroid approach is used, similarity can be measured in terms of angle θ given in Figure 3.34.

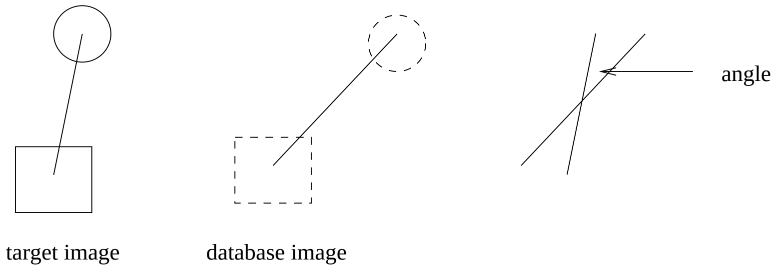


Figure 3.34: Match spatial relationship based on centroid

Similarity can be computed by taking $\mid cos\theta \mid$ or 1-$\mid sin\theta \mid$. In other words, as θ approaches zero, the similarity approaches 1.

To add the distance factor, the following general formula can be used:

$$similarity = w_d \mid d_{db} - d_{qry} \mid + w_\theta \mid cos\theta \mid, \tag{3.25}$$

where w_d and w_θ are the weights and w_d+w_θ=1. The distance between the objects, in the database image, is given by d_{db}, and the distance between the objects, in the target image, is given by d_{qry}.

In the current implementation, the Mbb approach is used. If two Mbbs satisfy a spatial relation a grade of 1 is assigned to the result; otherwise the grade is zero (refer Section 3.2.3).

Chapter 4

Query System

The DISIMA query system is featured by three components: the MOQL language, the parser and the engine. These components are explained in this chapter.

4.1 MOQL Language

MOQL is a multimedia query language based on ODMG's (Object Data Management Group) Object Query Language (OQL) [18]. MOQL is designed to supplement OQL by adding the expressive power to specify image queries. Furthermore, MOQL also defines the framework to capture the temporal element as well as some query functions in multimedia databases.

OQL follows the SELECT-FROM-WHERE syntax similar to SQL. Most of the extensions that MOQL introduces to OQL are in the WHERE clause [18]. These include the *spatial* and *contain* search conditions. The *spatial* conditions are constructed by using spatial objects (such as points, circles, line, etc.), spatial functions (such as length, area, etc.), and spatial predicates (such as cover, disjoint, etc.). A *spatial* condition can thus test whether a line intersects a point, a circle covers another circle, and so on. The *contain* condition has the basic form:

m `contains` o

where m is a media object, i.e. image, and o is the salient object (refer Section 3.2.2) in the media. The *contain* condition checks whether a salient object is in a particular media object.

The DISIMA project implements the *contain* and *spatial* definitions defined in MOQL, and expands the language by defining the shape, color and texture semantics.

4.2 A Query Parser for the MOQL Extension

SQL (Structural Query Language) is relational oriented and OQL (Object Query Language) is object oriented. Both languages lack the expressive power to describe image queries. The MOQL extension is designed to bridge the gap. In order to define queries on color, shape, texture and spatial relationship, new semantics and syntax are introduced in the MOQL extension.

Since MOQL queries follow the same SELECT-FROM-WHERE structure as traditional SQL queries, the design of the DISIMA parser is able to make use of the basic rules defined in SQL parsers. An example of these basic rules can be found in [17]. An extraction of the new rules defined on top of the basic rules is in Appendix B.

The objective of the DISIMA parser is to check the semantics and syntax of the external query, which is in the form of a character string, and convert the parsed string into an internal representation. Section 4.2.1 will explain the new semantics and syntax introduced in the MOQL extension, and Section 4.2.2 will discuss the internal structure representing a query.

4.2.1 New Semantics and Syntax

Query language needs to be user friendly so that users can simply describe "what" they want and not "how" to get the result. The following semantics and syntax are aimed to achieve this objective.

The *contains* condition

The result of traditional queries are often data values such as p.age, as in the following example:

```
SELECT p.age
FROM person p
WHERE p.lastname='Clinton';
```

In a MOQL image query the result is a set of images. The *contains* condition imposes a restriction on the images retrieved. The query

```
SELECT m
FROM image m, person p
WHERE m contains p
```

eliminates all images which do not contain persons.

Each *contains* condition names an object in an image. For example, if a query has two contains conditions, i.e.,

`m contains o1 and m contains o2,`

the resulting images contain at least object o1 and o2. However, if the conditions are connected by the operator *or* instead of *and,* i.e.,

`m contains o1 or m contains o2,`

the resulting images can contain only object o1, or only o2, or both.

The *shape* condition

If a condition requires keyword matching, e.g.,

`p.lastname='Clinton'`

the left-hand side of the comparison predicate specifies the object label and the attribute name. To distinguish *shape* matching from keyword matching, the reserved word *shape* is used. There are two types of shape conditions: general and specific.

- General shape condition–
  A general shape condition specifies a class name, e.g.
  `p.shape similar polygon`
  In this case, all Polygon class objects and objects belonging to the subclasses of Polygon–such as squares, rectangles and triangles–are retrieved. To retrieve squares and rectangles, the query is:

  ```
  SELECT m
  FROM image m, lso p
  WHERE m contains p
  AND p.shape similar rectangle.
  ```

  To obtain the same query result, the query can also be written as:

  ```
  SELECT m
  FROM image m, rectangle r
  WHERE m contains r;
  ```

To eliminate squares from rectangles, an additional restriction can be used, i.e.,

```
p.shape similar rectangle
and not p.shape similar square
```

- Specific shape condition–
  A specific shape condition specifies the dimension of the target shape, e.g.,
  `p.shape similar polygon(`$x_1, y_1 \ x_2, y_2 \ \ldots \ x_n, y_n$`).`
  The target polygon is defined by the list of points ($x_1, y_1 \ x_2, y_2 \ ... \ x_n, y_n$). The query result contains polygons which are similar to the target polygon, taking into account of the given coordinates.

The *color* condition

The color condition is written in the form:

`p.color similar colorgroup(`$r_1, g_1, b_1 \ r_2, g_2, b_2 \ \ldots \ r_n, g_n, b_n$`),`

where $n \geq 1$ and r, g, b are the red, green and blue values. An object in the database can be associated with n colors.

The *texture* condition

The *texture* condition is in the form:

`p.texture similar texturegroup(`$t_1 \ t_2 \ \ldots \ t_n$`),`

where $n \geq 1$ and t_i is a texture value measured in one of the n dimensions. An object can be associated with n texture values.

The *spatial* condition

As explained in Section 3.2, DISIMA uses Mbbs to estimate spatial relations–topological and directional. The general form of a spatial condition is as follows:

`p1.mbb relation p2.mbb,`

where "relation" can be *left, right, above, below, north, east, west, south, northeast, northwest, southeast, southwest, equal, cover, covered-by, inside, disjoint, overlap, touch, contain or overlapped-by.* A spatial condition can be expressed by using the inversed relation. For example, "inside" and "contain" are the inverse of each other and thus,

`p1.mbb inside p2.mbb`

produces the same result as

`p2.mbb contain p1.mbb.`

Note that "contain" (without "s") is a spatial relation, as distinguished from the "contains" in the *contains* condition.

The *and, or, not* conditions, and brackets

As with SQL and OQL, the operators *and, or, not* and brackets are used in MOQL queries. The operator "and" has precedence over "or" unless brackets are used to override the precedence. The syntax of a condition (search_condition) in a query is governed by a set of pre-defined rules, as shown in Appendix B. A search_condition can appear in the following formats:

- Predicate–
  A predicate can be a traditional comparison predicate, i.e., =, >, ≥, etc., or image predicate, i.e., contains and similar.
- NOT search_condition
- (search_condition)

- search_condition AND search_condition
- search_condition OR search_condition

Combining different search conditions with the operators, a complex query like:

```
SELECT m
FROM image m, building o
WHERE m contains o
AND o.shape similar polygon
AND NOT o.shape similar rectangle
AND (o.color similar colorgroup(200,200,200)
OR o.color similar colorgroup(0,255,0));
```

can be generated.

Subquery embedded in a condition

The DISIMA parser allows different ways to express a query, simple or nested. For example, a simple query like

```
SELECT m
FROM image m, polygon p, triangle t
WHERE m contains p
AND NOT m contains t;
```

can be expressed as:

```
SELECT m
FROM image m, lso o
WHERE m contains o
AND o.shape similar polygon
AND NOT o.shape similar triangle;
```

or can be written as:

```
SELECT m
FROM image m, lso o
WHERE m contains o
AND o not in  (SELECT m
               FROM image m, person WHERE m contains p);
```

The last one is a nested query which involves the execution of a subquery. Section 4.2.2 and 4.3 will explain how a subquery string is structured as a subtree and executed by the DISIMA engine.

Similarity required

As mentioned in Section 3.3, the similarity threshold can be defined at condition level, e.g.,

```
o.color similar colorgroup(200,200,200) similarity 0.8
```

or at global level, e.g.,

```
SELECT m
FROM image m, lso o
WHERE m contains o
AND o.shape similar circle
AND o.color similar colorgroup(200,200,200)
global similarity 0.8;
```

Number of images requested
To limit the number of images returned to the user, the *image_required* clause can be included in the query,

```
SELECT m
FROM image m, person p
WHERE m contains p
image_required 30;
```

4.2.2 Internal Structure of A Query

The input query to the DISIMA parser is a character string. After syntax checking, the query string is converted to an internal representation (*query* object) which can be executed by the query engine.

A *query* object stores all the information given by the query string. Its main components are the three pointers or oids (object identities) which can navigate to:

the select object which stores the information given in the SELECT clause.

the from object which stores the information given in the FROM clause.

the where object which stores the information given in the WHERE clause.

The similarity thresholds and number of images required, specified in the query, are also stored in the *query* object.

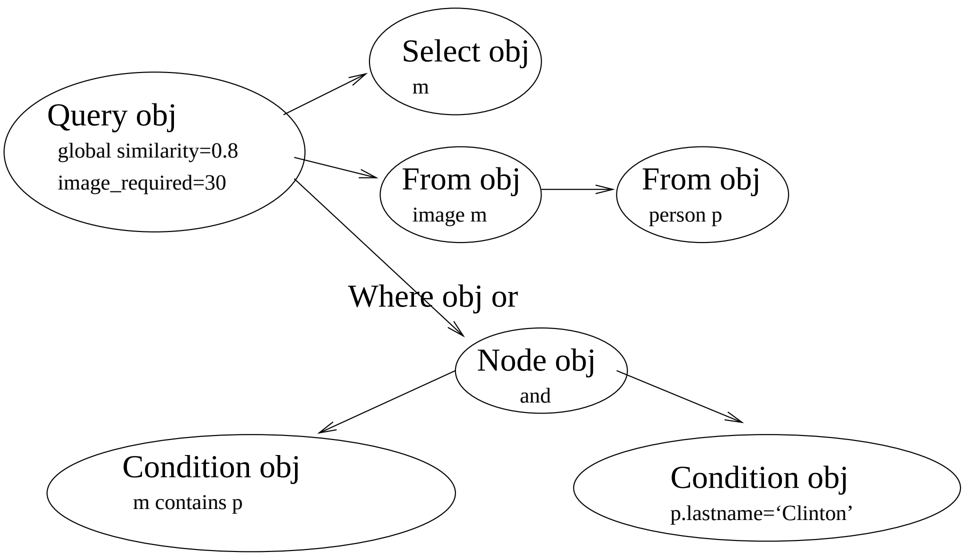


Figure 4.1: An example of a query object

The *query* object shown in Figure 4.1 corresponds to the query:

```
SELECT m
FROM image m, person p
WHERE m contains p
AND p.lastname='Clinton'
global similarity 0.8
image_required 30;
```

A query can have one or more *select* objects and *from* objects. In Figure 4.1 there are two *from* objects and one *select* object.

The *where* object (or *node* object) is the root of a tree structure. A query tree can contain a number of *node* and *condition* objects. A *node* object stores the operators *and* and *or,* and the result of the search conditions. The query tree built during the parsing stage is constructed following the order that search conditions are defined in the WHERE clause. Figure 4.2 explains how the *node* and *condition* objects are ordered in the query tree.

In Figure 4.2, *node* objects are drawn in solid line, and *condition* objects are drawn in dotted line. In each *node* object, the operator "or" can be used instead of the operator "and". "A", "B", "C" and "D" represent search conditions. A search condition can be a *comparison* predicate, i.e., $=$, $\geq$, $>$, $\leq$, $<$ and $<>$. For example,

```
o.lastname = 'Clinton'
```

It can also be a *contains* or *similar* predicate, or a *not in* predicate involving a subquery, e.g.,

```
m not in (SELECT m1
FROM image m1, person p
WHERE m1 contains p)
```

Figure 4.3 is a more complex example involving four levels of nodes and conditions. Figure 4.4 is an example of a condition involving a subquery.

The advantage of using a tree structure to store a query is that the nodes and leaves (conditions) can be rearranged to achieve query optimization similar to how *select, project* and *join* are manipulated in relational query trees.

4.3 A Query Engine for MOQL

After the parsing stage, a *query* object is ready for execution. It is now the responsibility of the query engine to execute the query and return the result to the user. Before explaining how the DISIMA engine works, the next section reviews some basic operations in relational query processing so that a comparison can be made between the relational and the DISIMA approach.

4.3.1 Relational approach

In the relational approach, each *entity* is defined by a *table.* The *attributes* of an entity are defined by the *columns* in the table. Tables can relate to each other through foreign keys. Figure 4.5 is an example of a relational database containing seven tables.

When executing a query, tables are *joined,* tuples are *selected* and attributes are *projected.* Relational query trees store the operators *select, project* and *join* in the nodes. Tables are stored in the leaves as shown in Figure 4.6.

The intermediate table after each join operation contains attributes required in the final result and also the foreign keys which are necessary for the join operations. These foreign keys may not be required in the final result. In contrast, the DISIMA query engine applies the OO *navigation* and *associative* access techniques. Only the oids of PSO objects are stored in the intermediate results, Other information on color, shape, texture, spatial, etc. can always be obtained through the PSOs.

4.3.2 DISIMA approach

In the DISIMA approach, entity and relation tables are replaced by objects, and instead of joining tables, related objects are located through pointers (or oids) as illustrated in Figure 4.7.

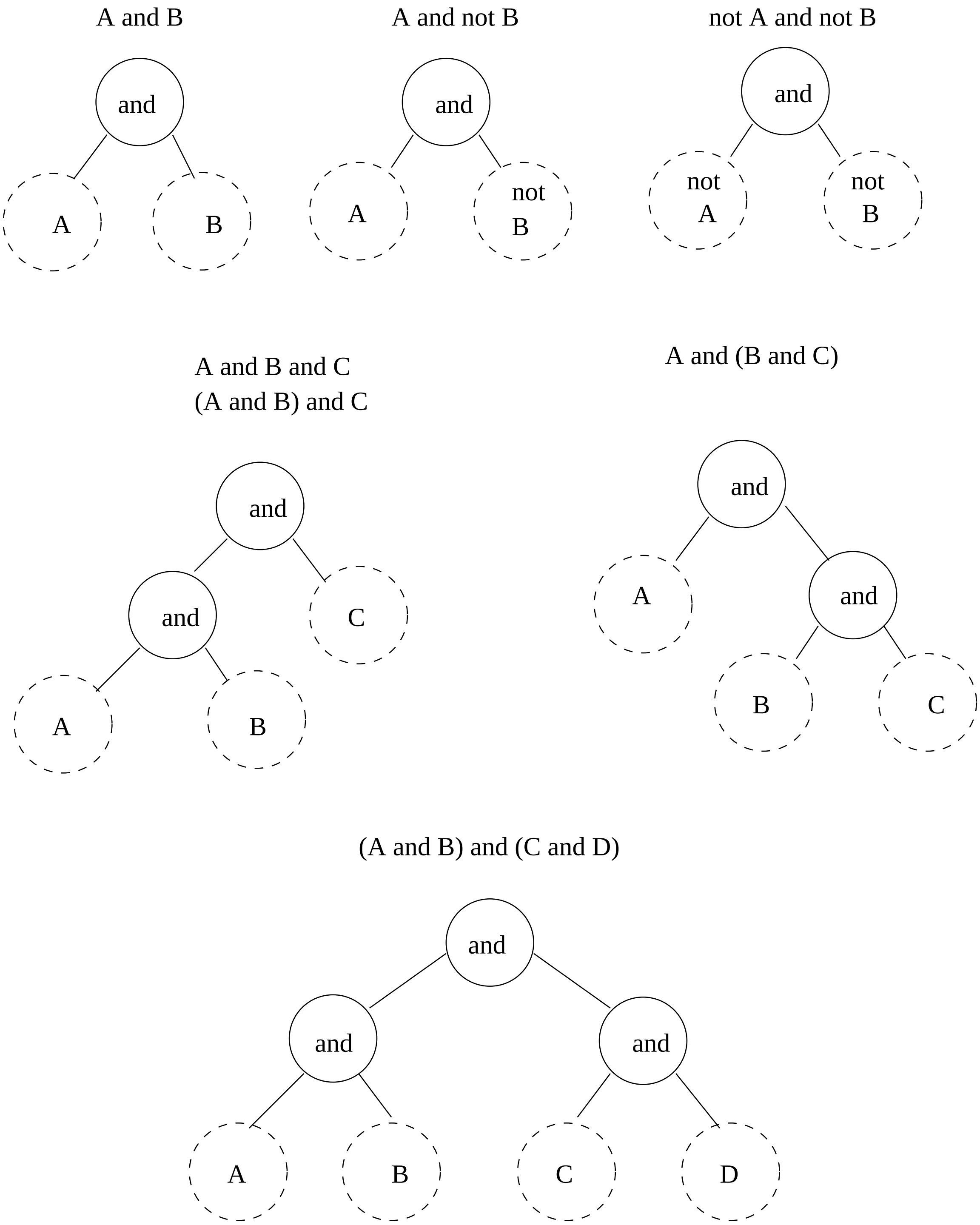


Figure 4.2: Examples of query trees

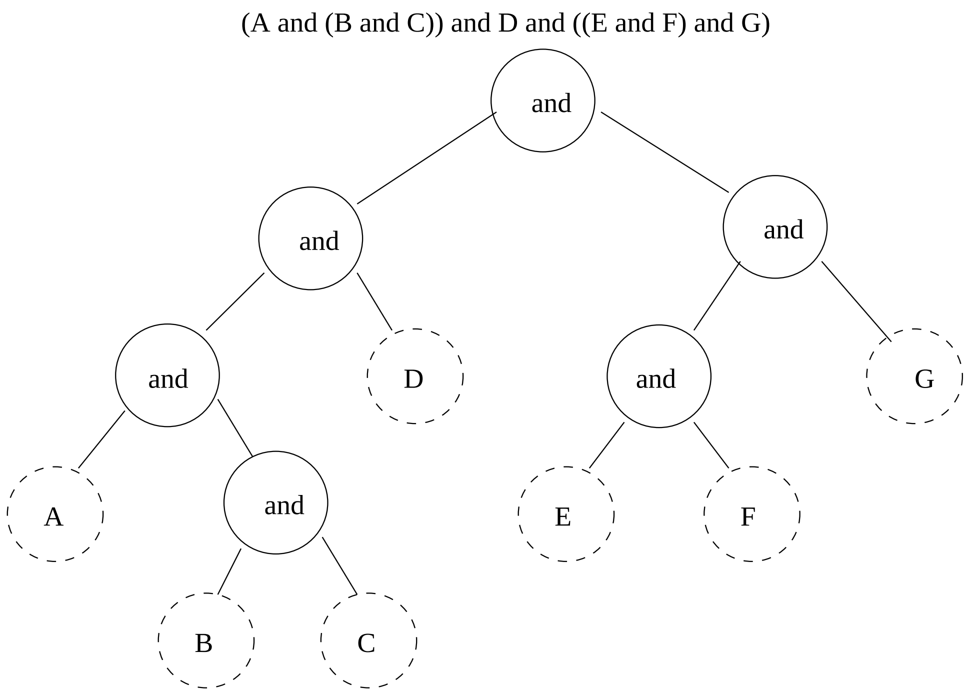
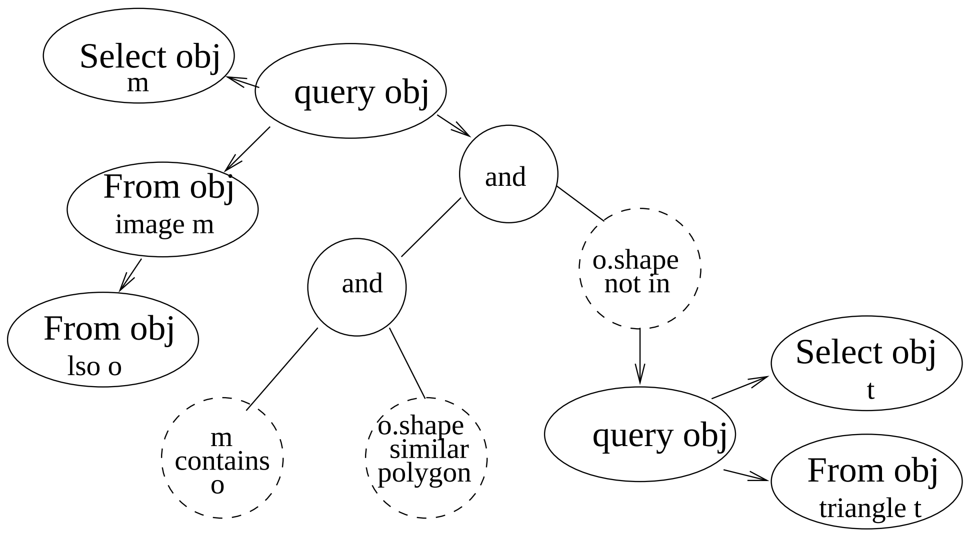


Figure 4.3: A query with a lot of brackets




```
select m
from image m, lso o
where m contains o
and o.shape similar polygon
and o.shape not in (select t from triangle t);
```

Figure 4.4: A nested query

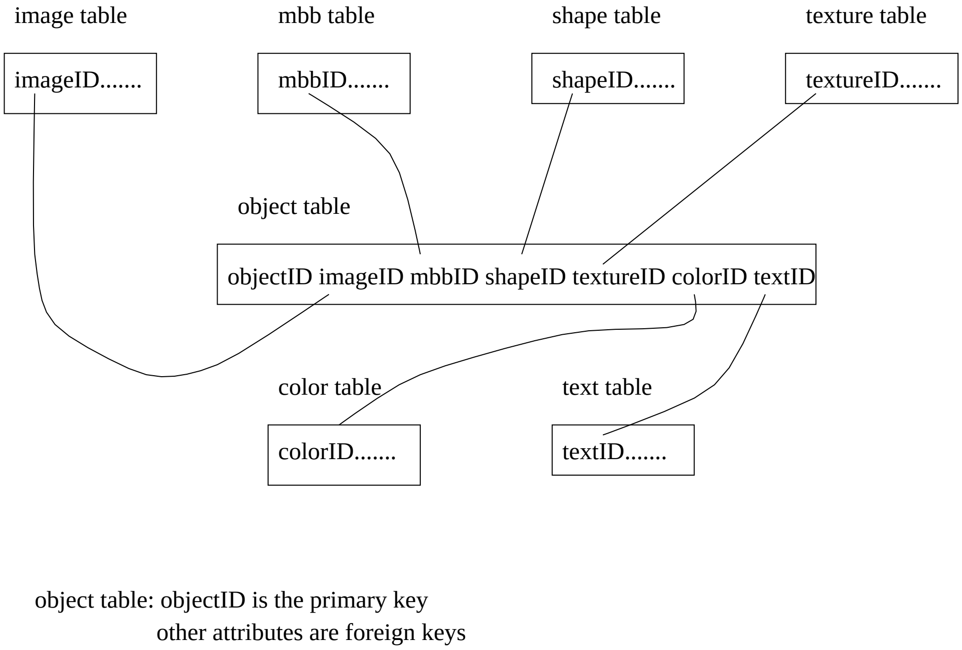


Figure 4.5: Relational entities and relationships

```
select m.author
from image m, object o, color c
where m.imageID=o.imageID
and o.colorID=c.colorID
and c.hue_value=250;
```

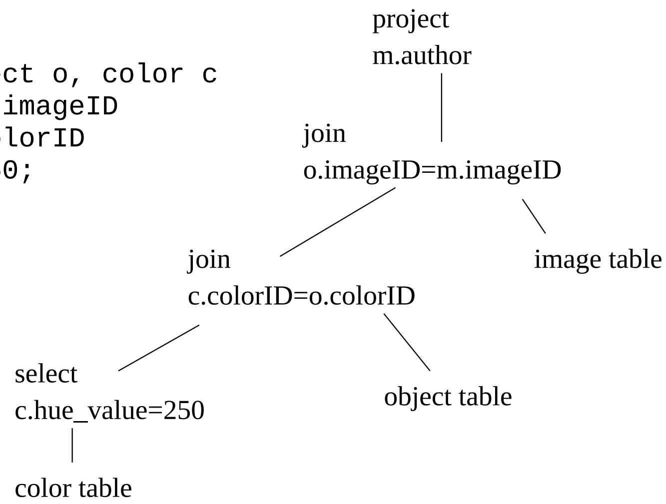


Figure 4.6: A relational execution tree

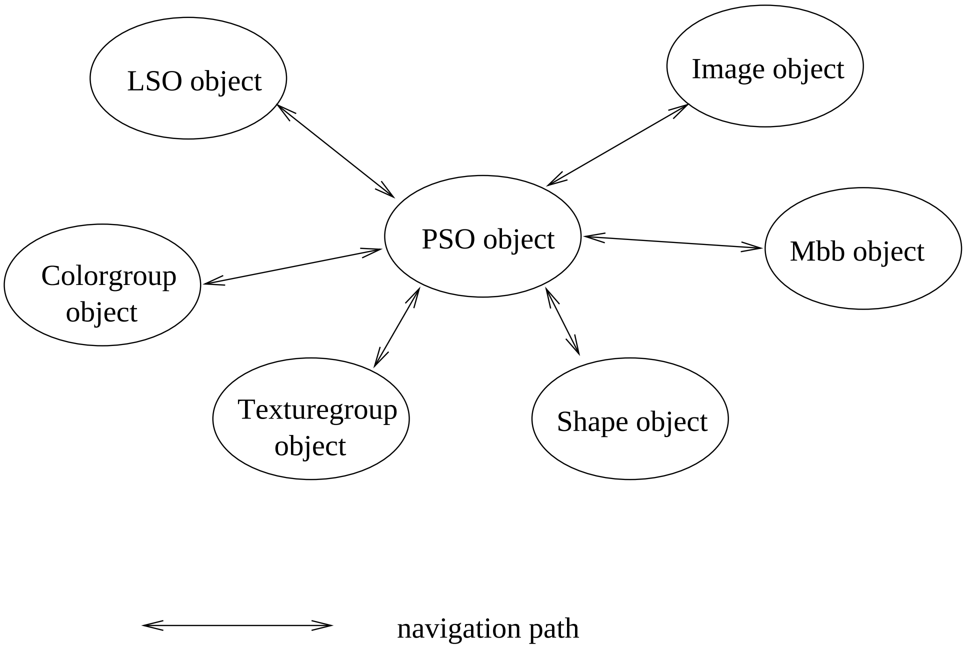


Figure 4.7: Navigation from the PSO object

In relational queries, the join-attributes have to be specified explicitly, e.g., o.key = p.key, but in MOQL queries, navigation through oids is implicit and does not need to be specified in the query. As mentioned in Section 3.2, the bridge that allows navigation is provided by the PSO objects. From a PSO object, the associated color, shape, texture, spatial relation, image and LSO data can be obtained. The DISIMA execution tree differs from the relation execution tree in the following ways:

- Class *extents* and not tables are stored in the leaves.
- Instead of the *join* operator, the DISIMA tree stores the operator *and (intersect)* and *or (union)* in the nodes.
- The *selection* and *projection* operations are applied to the class extents and not the tables.
- The intermediate result identifies a set of PSO objects satisfying the conditions.

The rest of this chapter will explain the naming convention used by the DISIMA engine, how a query is executed at condition and node levels, and then discuss the structure used to store the query result.

Naming Convention used by the DISIMA engine

In traditional SQL queries, each selected item is considered separately, and the items are not viewed as objects in images. For example, the SQL query:

```
SELECT p1.age, p2.age
FROM person p1, person p2
WHERE p1.lastname='Clinton'
AND p2.firstname='Bill';
```

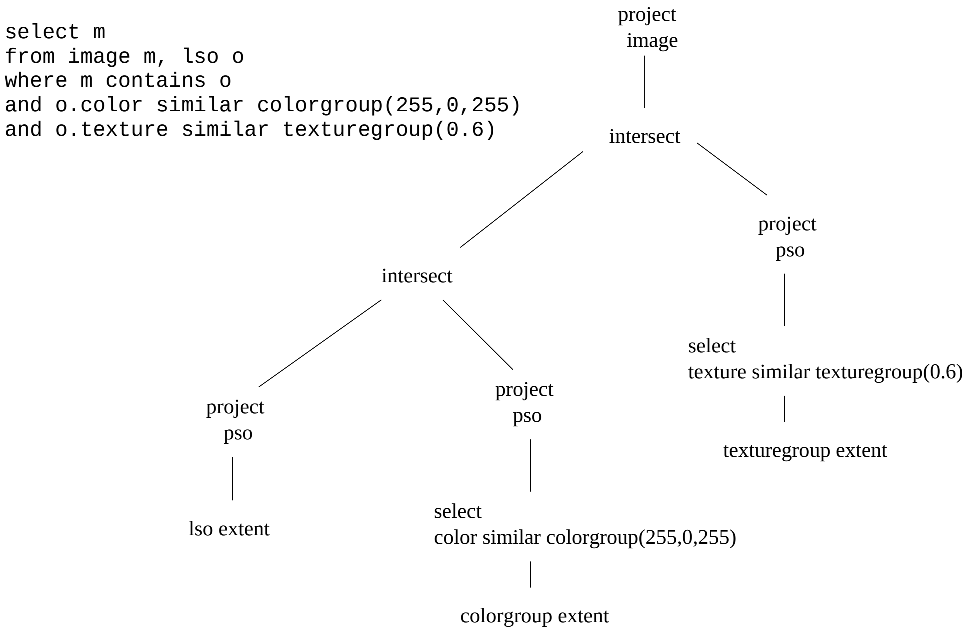


Figure 4.8: DISIMA execution tree

will select the age of any person who has either the lastname equal to "Clinton", or the firstname equal to "Bill". A person called "Bill Clinton" satisfying both conditions therefore has his age selected twice, i.e., p1.age and p2.age. In other words, by default, p1 and p2 can refer to the same entity.

Since spatial relation is one of the features introduced in the MOQL extension, the naming convention of the DISIMA engine is different from that of the SQL engine.

Different labels represent different objects in an image

In the DISIMA query:

```
SELECT m
FROM image m, person p1, person p2
WHERE m contains p1
AND m contains p2
AND p1.mbb above p2.mbb;
```

by default, p1 and p2 refer to two different PSO objects in an image. The selected images should contain two persons (at least) with one above the other. Using this naming convention, the query:

```
SELECT m
FROM image m, person p1, person p2
WHERE (m contains p1
AND p1.lastname='Clinton')
AND (m contains p2
AND p2.lastname='Clinton');
```

will retrieve images containing at least two persons whose lastnames are both "Clinton", e.g., Bill Clinton and Hillary Clinton. Images containing only one "Clinton" will not be retrieved. The default can be overriden by adding the condition "p1 = p2". If so, images containing one or more "Clinton" will be retrieved. If the query is:

```
SELECT m
FROM image m, person p1, person p2
WHERE (m contains p1
AND p1.lastname='Clinton')
AND (m contains p2
AND p2.firstname='Bill');
```

the selected images will contain at least two persons. Images containing "Bill Clinton" alone (without another person with lastname "Clinton", or without another person with firstname "Bill") will not be retrieved.

Execution at Condition Level

The execution plan shown in Figure 4.8 has three conditions:

```
m contains o
o.color similar colorgroup(255,0,255) and
o.texture similar texturegroup(0.6)
```

The class extents being searched are the LSO, colorgroup and texturegroup extents. All the objects in the LSO extent satisfy the contains condition. Only objects which have color similar to colorgroup(255,0,255) are selected as the result of the color condition. Similarly, only objects which have texture similar to texturegroup(0.6) are selected as the result of the texture condition. The result of each condition is stored in a *resultpso* object. The *resultpso* object carries the label *o* specified by the condition, and a set of *unitpso* objects which identifies the PSOs satisfying the condition. The *resultpso* object and *unitpso* objects are illustrated in Figure 4.9.

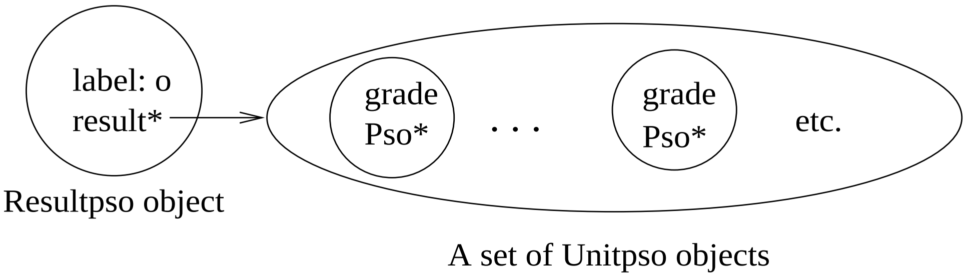


Figure 4.9: The result of a condition: The Resultpso and Unitpso objects

An *unitpso* object carries two pieces of information: the oid of the PSO selected by the condition, and the grade awarded to the PSO computed based on a distance function discussed in Section 3.3.

An alternative approach to process the *contains* statement

Like the *similar* statement, the *contains* statement is treated as a search condition in the current implementation. Due to the special semantics of *contains,* another approach is to process it as a *join* similar to the relational join. Consider the following two queries:

```
SELECT m
FROM image m;
```

and

```
SELECT m
FROM image m, person p;
```

Suppose there are n PSOs in all the images and only PSO_{11}, PSO_{12} and PSO_{13} are person objects. In both queries, the result is a set of images. Although person is mentioned in the second query, it has no effect on the query result because the two sets of PSOs are not joined (Figure 4.10 (a)). By adding the *contains* statement, the second query becomes:

```
SELECT m
FROM image m, person p
WHERE m contains p;
```

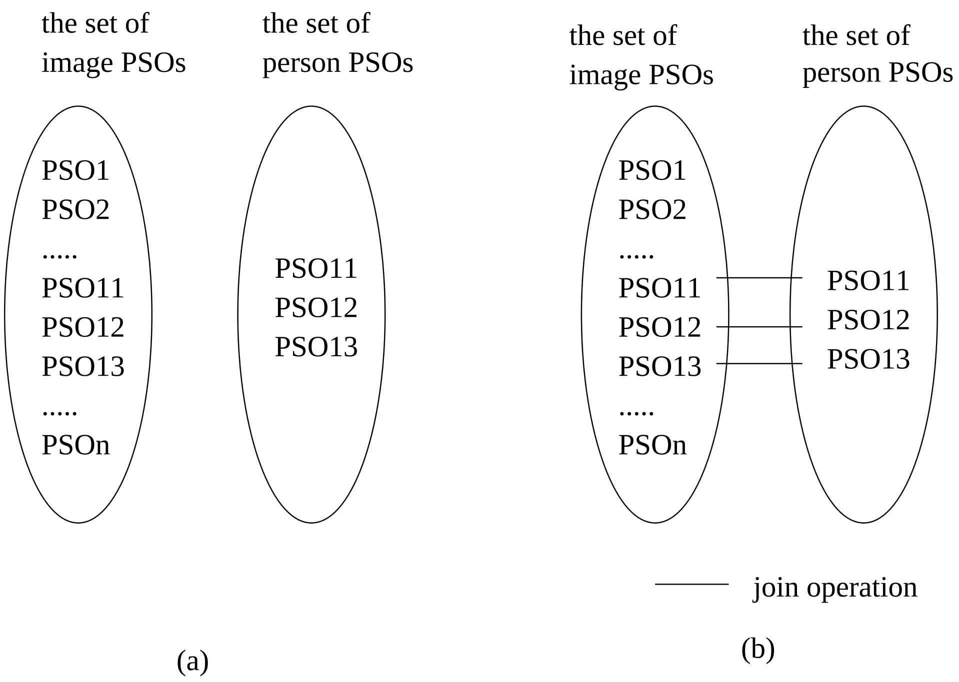


Figure 4.10: To process "contains" as a join

Figure 4.10 (a) shows that without the contains statement, the query selects all the images. The person PSOs constitute a subset of the image PSOs. When the image PSOs are joined with the person PSOs, Figure 4.10 (b) shows that only the intersection (images containing PSO_{11}, PSO_{12} and PSO_{13}) are selected.

Whether navigational access, using oids to process the contains statement, is more efficient than the join operation is debatable. It was pointed out that the OO approach provides alternative execution plans that can be considered in addition to using join [4]. Since optimization is not the focus of this report, readers interested in the topic can refer to more specific research papers [4].

Execution at Node Level

In a query tree, each node is associated with either one or two conditions. When there are two conditions, either intersection or union is taken from the results of conditions, depending on the operator.

When the operator is *and,* intersection is taken. However, two *resultpso* objects can intersect only if they have the same label, i.e., the conditions apply to the same object. If the labels are different such as that in the following conditions:

`o.color similar colorgroup(255,0,255)` and
`p.texture similar texturegroup(0.6),`

the image is expected to contain at least two objects, with one having color similar to colorgroup(255,0,255) and the other having texture similar to texturegroup(0.6). In this case, both *resultpso* objects, instead of their intersection, are stored in the node as *resultset* object. Figure 4.11 (a) and (b) show that after the *and* operation the *resultset* object can contain one or two *resultpso* objects depending on whether their labels are the same.

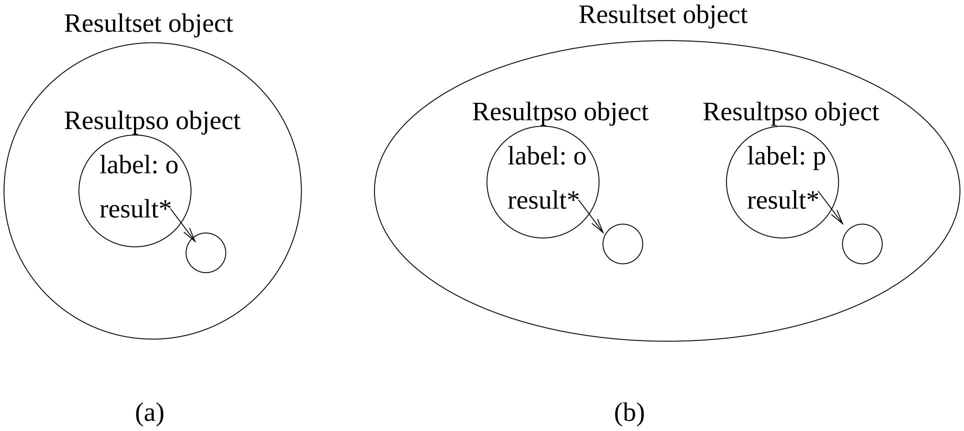


Figure 4.11: The result of the *and* operator

In the example query used in Figure 4.9, since both the color and texture conditions apply to the same object *o*, the *resultset* object contains only one *resultpso* object – the intersection (Figure 4.11 (a)). Objects which satisfy only one condition are eliminated. When an object satisfies both conditions, the final grade is the average of the grades awarded from individual conditions.

If the operator is *or* instead of *and*, the *union* of the *resultpso* objects is taken. Suppose the two *resultset* objects shown in Figure 4.11 (a) and (b) are processed by the operator *or*, the union is a set of *resultset* objects shown in Figure 4.12.

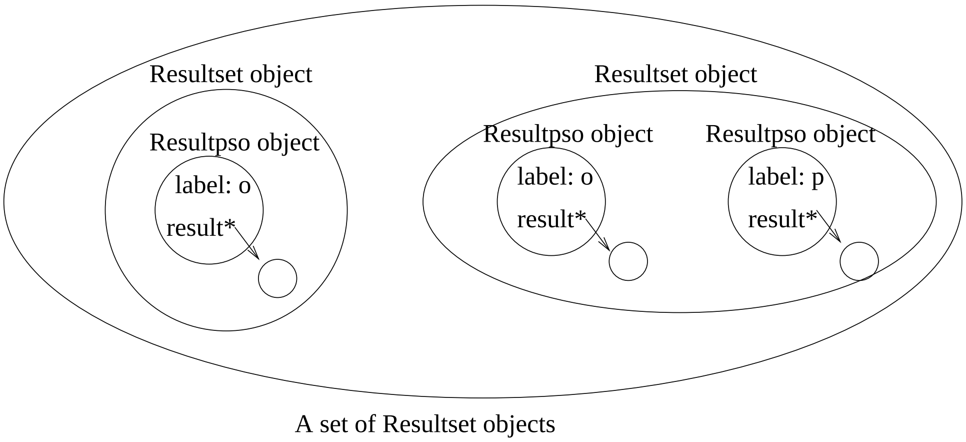


Figure 4.12: The result of the *or* operator

How to execute the operator *not*

The *not* operator in a query can appear at two levels: condition level and node level.

“Not” at conditon level

In relational queries, *not* is seldom used except in the *not in* statement because the comparison predicate $=, <>, \geq, <, \leq$ and $>$ can often do the job. For example,

```
not o.lastname = ‘Clinton’
```

can be expressed as,

```
o.lastname <> ‘Clinton’
```

However, the MOQL extension has new predicates *similar* and *contains*, which do not have an inverse. The use of *not* is therefore necessary to retrieve the complement set of objects, i.e.,

```
not o.color similar colorgroup(255,0,255)
```

How to set the search boundary for the operator “not”

Each condition is executed on a class extent and therefore the search set is bounded by the set of objects in the extent. When *not* is added in front of the condition, there are two possibilities. One is like the *not*-color condition above. In this case the search set is still bounded by the set of objects in the colorgroup extent; any object which is similar to colorgroup(255,0,255) is selected while the other objects in the extent are disregarded. A different *not* operation is illustrated by the following query:

```
SELECT m
FROM image m, person p
WHERE not m contains p;
```

The search set of the condition is the person extent but the *not* operator requests objects outside the person extent. To overcome this problem, the concept of *closure* is required so that the search space of each condition, with or without the *not* operator, is bounded.

What is a bounding set and a universal set

Since PSO objects satisfying the conditions are selected and stored in the intermediate results, the set of PSOs represents the entire search space. The *universal set* is defined as the set of all PSOs in the database. A bounding set in the context of DISIMA is defined as the union of set_A and set_B such that, set_A is the set which satisfies the non-not condition and set_B is the set which satisfies the not condition. The relation between the universal set and a bounding set is:

$$boundingset \subseteq universalset \tag{4.1}$$

The universal set contains PSOs appearing in all images. Images may include medical_images, scene_images, painting_images and so on. In the not-contains query above, the condition result set_B is obtained by subtracting set_A from the bounding set. In this example, the bounding set is the same as the universal set because the query uses m (images) in the select clause. If the query selects *medical_image*, the universal set is still the set of all PSOs but the bounding set is the set containing only PSOs appearing in medical_images.

The bounding set is therefore governed by the select clause in the query. The search space of a condition is closed by the bounding set.

“Not” at node level

To handle *not* at node level involves propagating the operator to the subtree. An algebraic expression can explain this concept clearly. Let the query result shown in Figure 4.12 be written in the algebraic form:

(R_{cond1}) or $(R_{cond2}$ and $R_{cond3})$

where R_{condi} is the result of condition i. If *not* is applied, i.e.,

$\sim ((R_{cond1})$ or $(R_{cond2}$ and $R_{cond3}))$
after propagation using De Morgan's rule, the algebraic equation becomes:
$(\sim R_{cond1})$ and $(\sim R_{cond2}$ or $\sim R_{cond3})$

A similar concept is used by the DISIMA parser and query engine. When the parser detects a *not* at node level, it will call the function propagate_not() which will pass the *not* to each condition in the subnodes. If the condition has a *not* already, the two nots will cancel out. Any *and* operator in the subnodes will be changed to *or* and vice versa.

Processing the query tree

Once the query tree is constructed, the query engine initiates post-order traversal. The left-condition is executed first, before the right-condition, and any subnode is executed before the higher-level node.

The execution plan used in this report is based on the query tree translated directly from the query string. Optimization can be achieved by manipulating the nodes and leaves of the tree. More about optimization can be found in Section 6.2.

Storage of temporary results

While the query object stores all the information of a query string, the result object stores the information necessary for the retrieval of final images. The intermediate result after execution at condition or node level is defined by a set of *resultset* objects. The *resultset* objects are constructed using the concept of disjunctive normal form as illustrated in Figure 4.13.
Take a simple query:

```
SELECT m
FROM image m, person p
WHERE m contains p;
```

which has only one condition. The result (a set of *resultset* objects) is shown in Figure 4.14 (a). The *label* associated with the *resultpso* object is *p* . If the query has two or more conditions connected by the operator *and,* and all the *resultpso* objects have the same label, the result still contains only one *resultpso* object. The pointer (result*) navigates to a set of *Unitpso* objects. Each *Unitpso* object has the grade information and the oid of the PSO object which satisfies the condition(s).
If a query specifies more than one object, e.g.,

```
SELECT m
FROM image m, person p1, person p2
WHERE m contains p1
AND m contains p2
AND p1.lastname='Clinton';
```

the *resultset* will have more than one *resultpso* object as shown in Figure 4.14 (b). The *resultpso* objects of different labels represent different objects in the image. In this example, resultpso1 is labelled p1 (object1), resultpso2 is labelled p2 (object2) and resultpso3 is labelled p1 (object1). That means two restrictions are imposed on object1, i.e., object1 is a person and object1.lastname='Clinton', and only one restriction is imposed on object2, i.e., object2 is a person.

Instead of the second *and* operator, assume it is replaced by the *or* operator. The result is shown in Figure 4.14 (c). Although both resultpso1 and resultpso3 have the same label *p1,* the two sets of unitpso objects are not intersected because the query requires that either object1 is a person or object1's lastname = 'Clinton'. Intersecting will incorrectly eliminate the person objects whose name is not 'Clinton'.

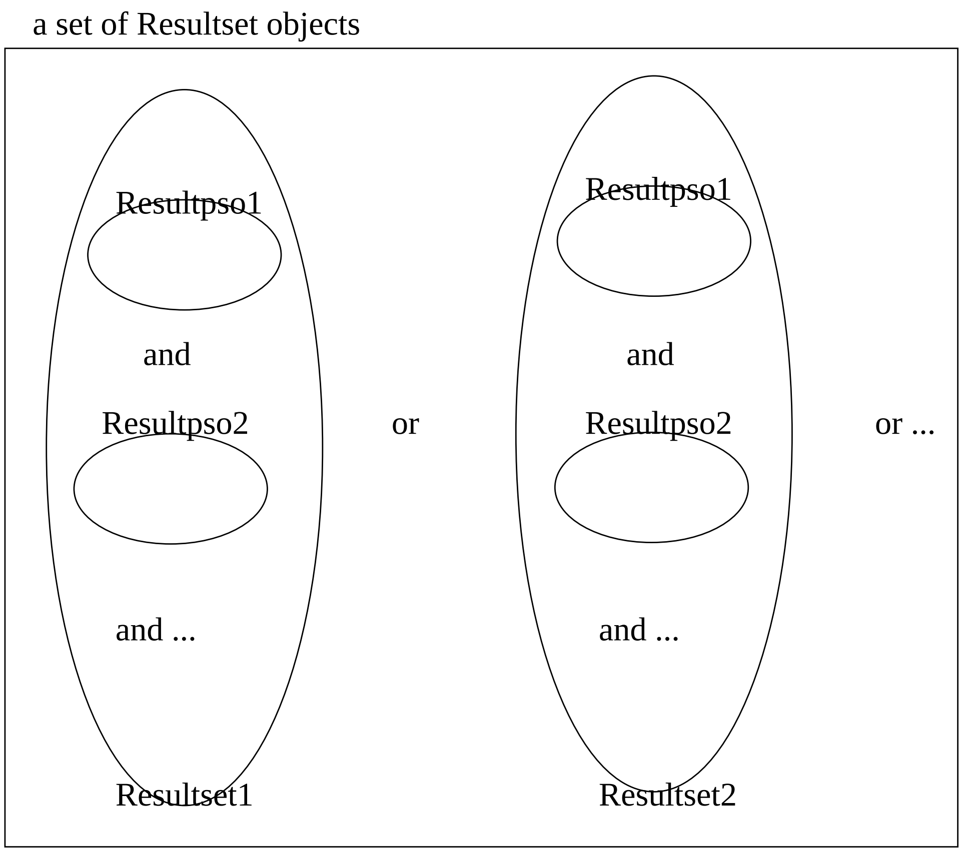


Figure 4.13: Temporary result illustrated in disjunctive normal form

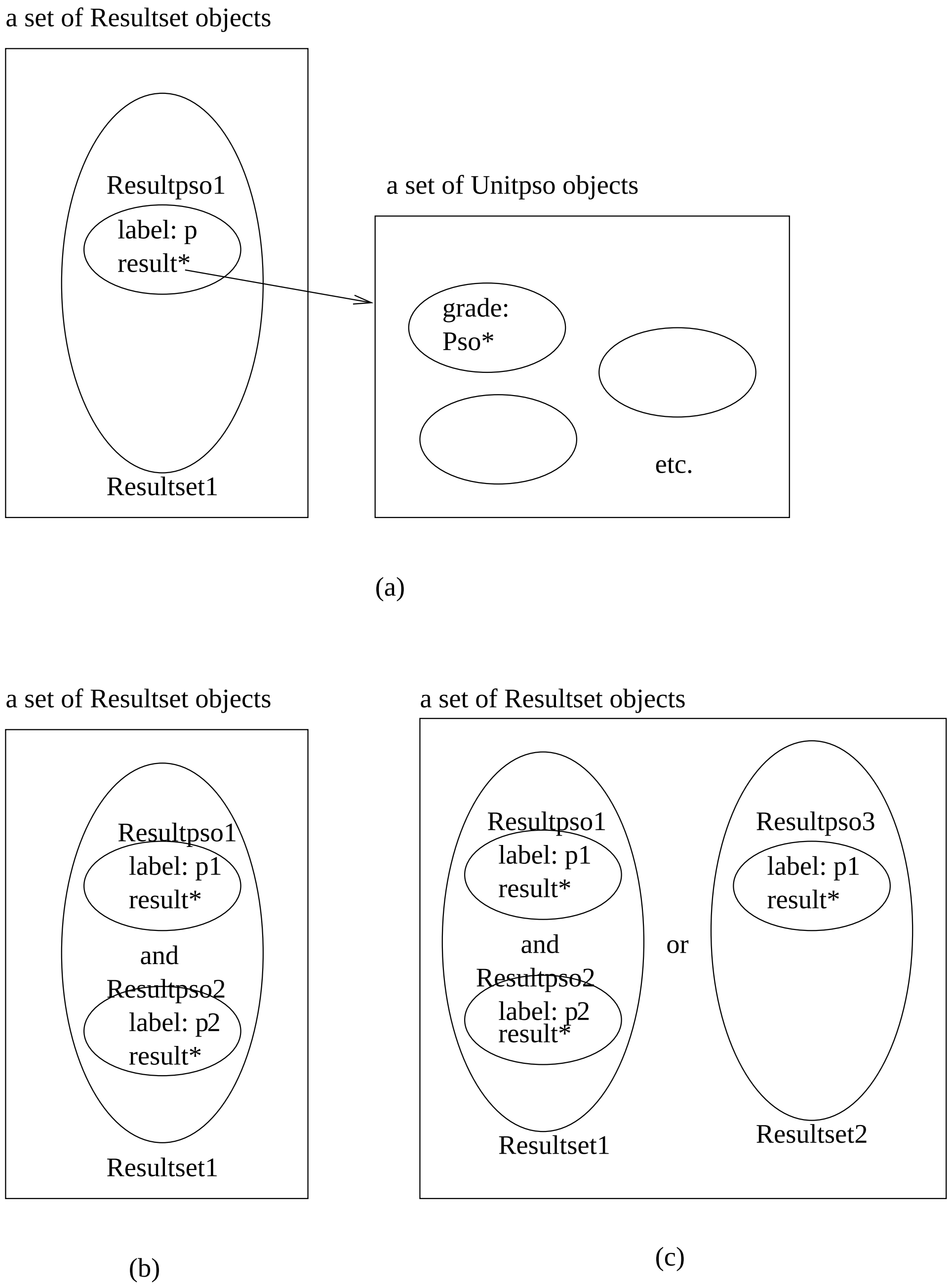


Figure 4.14: Examples of temporary result

Another way to store the result is to follow conjunctive normal form so that:

$Resultpso_{1\cup 3}$ and ($Resultpso_2$ or $Resultpso_3$)

is used instead of

($Resultpso_1$ and $Resultpso_2$) or $Resultpso_3$.

The disadvantage of using conjunctive normal form is that each "or" group, e.g.,

($Resultpso_2$ or $Resultpso_3$)

does not give the final images, because there are unprocessed "and" operator. On the other hand, each "and" group, e.g.,

($Resultpso_1$ and $Resultpso_2$)

contains the information of the final images; the unprocessed "or" operator will add more images, but will not disqualify those obtained from the "and" group.

$Resultset_i$ in Figure 4.14 is implemented as a *resultset* object. Each *resultset* object is in turn defined by a set of *resultpso* objects. A *resultset* object is added to the result when the *or* operator is encountered, while a *resultpso* object is added to the *resultset* when the *and* operator is encountered.

<u>Apply distributive rule to maintain disjunctive normal from</u>

When executing complex queries, it is necessary to combine two intermediate results which may contain a few *resultset* objects and a number of *resultpso* objects within a *resultset.* In such cases, the query engine will apply the distributive rule. Suppose the two intermediate results are written in the following algebraic form:

$$(Resultpso_1 and Resultpso_2) or (Resultpso_3 and Resultpso_4) \tag{4.2}$$

and

$$(Resultpso_5 and Resultpso_6) or (Resultpso_7 and Resultpso_8). \tag{4.3}$$

After applying the distributive rule, the algebraic equation becomes,

$(Resultpso_1 and Resultpso_2 and Resultpso_5 and Resultpso_6)$

$or (Resultpso_3 and Resultpso_4 and Resultpso_5 and Resultpso_6)$

$or (Resultpso_1 and Resultpso_2 and Resultpso_7 and Resultpso_8)$

$or (Resultpso_3 and Resultpso_4 and Resultpso_7 and Resultpso_8)$.

Each conjunction group corresponds to a *resultset* object. The result is a set of *resultset* objects.

Final result: retrieve images based on the resultset objects

Since the oids of the PSOs satisfying the query are stored in the *resultset* objects, the oids of the images containing these PSOs can be obtained (Section 3.2). The reason why the result is stored in disjunctive normal form and not conjunctive normal form is now clear. If it is in disjunctive normal form, the images retrieved by scanning each *resultset* can be returned to the user without further processing. On the other hand, if it is in conjunctive normal form the images obtained from each *resultset* object are not final because there are unprocessed *and* operations which may disqualify some of the images.

Synchronization between different resultpso objects within the resultset object

The *synchronize() function* is designed to put a final check on each *resultset* object before the images are extracted and returned to the user.

A *resultset* object may contain n *resultpso* objects of different labels meaning that the image should contain at least n objects. The *synchronize() function* ensures that images containing fewer than n objects will not be retrieved. The similarity grade assigned to the retrieved image is the average of the grades scored by the n *unitpso* objects.

How to handle result of subquery?

As with blocks defined in programming languages, a query language also has the concept of block–but in this case, a *subquery block.* A label defined within a subquery is visible only to the subquery and not to the master query. For example,

```
SELECT m
FROM image m, person p
WHERE m contains p
AND p not in (SELECT m1
              FROM image m1, person p1
              WHERE m1 contains p1
              AND p1.lastname='Clinton');
```

The labels m1 and p1 are visible only within the subquery. After the subquery is processed, the result or the set of objects is not associated with the label p1 or m1 but p, which is a label in the master query. The query engine converts the labels of the *resultpso* objects when exiting a subquery, ensuring labelling consistency.

Chapter 5

Implementation and Limitations

5.1 Introduction

The design of the DISIMA type system, query parser, and engine has been discussed in previous chapters. In order to illustrate the concept and show how the system works on live data, a demo database containing more than one hundred images and thumbnails was constructed. MOQL queries can be input as a character string or through a graphical user interface [29] at the site:

`http://www.cs.ualberta.ca/~database/IDB/Interface.html.`

Since the focus of this report is not research on image processing, there is no detailed analysis and evaluation on the various image processing algorithms. It is up to individual applications to decide which algorithms meet their requirements. In order to show how these algorithms can be integrated with the type system, some generally used distance functions are taken from the literature and implemented in this report. Other distance functions can be used by modifying the *query()* and *equal()* function in the type system.

The type system and query components are built on top of ObjectStore version 5.0, which provides basic database facilities such as storage management, recovery, and transaction control. The development environment used in this report also includes C/C++, Flex/Yacc and Perl.

Implementation of the type system was demonstrated in Italy in May 1998, and some users in Germany have been experimenting with it since then. In addition to the demo database, an application is being developed in collaboration with Photo Services of the University of Alberta. The intention of this project is to computerize their photo collection which includes convocations, celebrations, historic scenes, artifacts, and research activities of importance to the university.

5.1.1 Why ObjectStore?

A major advantage of ObjectStore is its close integration with the C^{++} language, and its persistent storage capabilities for C^{++} objects. Traditionally, application developers using C^{++} or C have been responsible for writing detailed code to create, access, and update persistent values, translating the disk representation of an object to the representation used during execution. By using *new()* and a transaction boundary, ObjectStore persistence is transparent to the programmer. Since transient and persistent objects have the same representation, type checking is possible on ObjectStore program variables. This avoids the *impedance mismatch* problem between a database system and its programming language, where the structures provided by the database system are distinct from those provided by the programming language.

ObjectStore allows a relationship to be set up between two objects, which can be 1 to m, m to 1, or m to n. When a link is established in one direction, the other direction is automatically set up. For example, if the application program writes a_pso→shape.insert(a_shape), then a_shape→pso.insert(a_pso) is set up by ObjectStore. When one link is deleted, its counterpart is deleted automatically. This is particularly useful when setting up the links between the PSO instances and associated instances, i.e., color, shape, texture, Mbb, and image.

The Collection class in ObjectStore allows programmers to define different structures: array, list, set, and bag, by calling some built-in functions. The elements in the collection can be ordered or unordered, with duplicate or without, etc., as specified by the programmer. Complex collections can be created with minimal effort.

Another reason for using ObjectStore is because an early project–A Generic Type System for a Multimedia Database System–was also built on top of ObjectStore. Using the same underlying DBMS facilitates integration of the two projects.

5.1.2 Why Flex and Yacc?

Lex and Yacc perform the work of a lexer and scanner on structured text. Flex is a freely available version of lex. Since the MOQL extension is a newly defined language [18], there is no current parser that can be used. The responsibilities of the DISIMA parser are to check the syntax of the query string and construct the query tree which can be executed by the query engine. The parser can be written in C/C^{++} or Flex/Yacc. The latter was chosen because it requires less code and thus is easier to debug, and provides a systematic way for syntax checking and query tree construction.

The two tasks that need to be performed repeatedly on the input query string are: dividing the input into meaningful units, and discovering the relationship among the units. Flex/Yacc allows *tokens* to be constructed and related to each other through a set of rules defined by the programmer. Once the syntax is validated, objects can be constructed and inserted into the query tree based on the parsed data. The tokens are defined in the file scan.l and the rules in moql.y.

5.2 How are the programs organized?

The programs are organized in different directories, as shown in Figure 5.1.

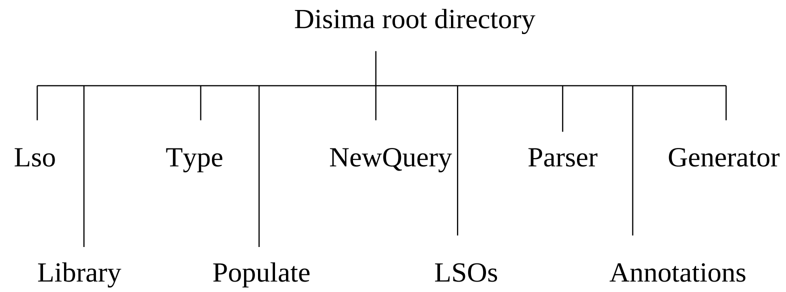


Figure 5.1: The DISIMA directories

The Type, Parser, NewQuery, and Generator directories store the code of the type system, the parser, the query engine, and the schema generator, respectively. LSO is the root of all LSO classes, and is defined in the Type directory. The other user-defined LSO classes are defined in the Lso directory.

The DISIMA program to populate the database is found in the Populate directory. Data files in the Annotations and LSOs directories are read during database population.

Three shared libraries: libLSO, libQry, and libType, are created during compilation and are stored in the Library directory.

Other technical details can be found in the README file in each directory, and the comment beside the code.

5.3 Schema generation

An integral part of the type system is the LSO class hierarchy, which is application dependent and user defined. The LSO classes used in the demo database are shown in Appendix C. No matter which LSO classes are used by an application, the common requirement is to allow a class to be added, deleted, or modified. For example, a user may want to delete the *Office* class from the demo hierarchy or to add a new *Tower* class, as a subclass of *Building.* To eliminate the need for the user to learn how to define the methods of the new class, and modify the code to accommodate the change, a schema generator is implemented to automatically generate all the necessary code to be complied with the rest of the system. What is required from the user is simply to specify the new class name, its super class, and its attributes, according to the format shown in Appendix C–or to delete the class information which is no longer required. The user-defined LSO class information is stored in the file UserClasses.

When the code is recompiled by typing *make* at the DISIMA top directory, the generator is called automatically. The files that will be regenerated, based on the modified UserClasses file, are: lsoClasses.hh, lsoClasses.C, schema.C for the Lso directory, initializeLso.C, createLso.C, containLso.C and conditionLso.C. The old version of these files is saved before it is overwritten by the newly generated version. In case the compilation is not successful, the user can always revert to the old version.

5.4 Data population

Data population is carried out in two stages: the populateLso stage for LSO objects, and the populate stage for other objects. PopulateLso uses the annotation files in the LSOs directory to create all the necessary LSO objects in the database. PopulateLso is implemented by another project member.

Figure 5.2: An image containing five golden fish as salient objects

Included in the work of this report is the population of the image, PSO, shape, color, texture, and Mbb objects. The populate program uses the files stored in the Annotation directory. Each annotation file contains data of an image. For example, the image containing five gold fish (Figure 5.2) has an annotation file *golden_fish.ant,* providing the following information:

```
Fish 98
MBB (393 79) 458 112
```

```
Shape rect[(393 79) 458 112]
Color (250 116 31)

Fish 98
MBB (431 232) 229 265
Shape rect[(431 232) 229 265]
Color (251 254 137)

Fish 98
MBB (640 403) 297 226
Shape rect[(640 403) 297 226]
Color (253 103 29)

Fish 98
MBB (325 390) 117 224
Shape rect[(325 390) 117 224]
Color (255 255 255)

Fish 98
MBB (267 176) 130 243
Shape rect[(267 176) 130 243]
Color (249 0 58)
```

Each fish (PSO) in the image has its own Mbb, shape, and color. *Fish* is the class name, and 98 is the LSO logical id which is used to retrieve the other LSO information–such as comName and sciName, already populated in the database during the populateLso stage.

In the demo database, texture data are not yet populated. The populated shapes include rectangle, square, circle, and ellipse. Although the type system can associate a set of colors with an object, only the average color is stored currently. More data will be populated into the database when they are available.

5.5 Database images vs. file system images

There are two ways to store images. One way is to store them in the file system, and the other is to store them in the database. Storing data in the database has the advantage of making use of the underlying facilities such as recovery, transaction, and concurrency control, ensuring consistency in the database.

The DISIMA type system supports the database approach by storing the raw images and thumbnails in the image objects. The data are stored as byte streams. These byte streams will be supplied to ObjectForm and displayed to the user. ObjectForm is the graphical interface associated with ObjectStore, and will act as the interpreter between the current MOQL graphical interface and the query engine. Since the MOQL interface is implemented in Java and cannot access the image objects in ObjectStore, ObjectForm will transform the query result into a HTML format which is then returned to the MOQL interface.

Before ObjectForm is fully integrated with the DISIMA components, demo images are stored in the file system. The access paths to the image and thumbnail are stored in the image object. When the set of qualified image oids is returned by the engine, the function *print_html(),* defined in scan.l, will retrieve and embed the paths in a HTML document, which is then returned to the MOQL interface. Currently, the access paths for images and thumbnails are:

iglinski/Disima/Images/ and

iglinski/Disima/Thumbnails/
respectively.

5.6 Similarity match

Since the choice of distance functions is application dependent, the functions, i.e., the turning anlge function, which have been implemented in the DISIMA type system serve only as an example of how similarity match can be performed by the query engine. Further analysis and evaluations have to be done in the image processing area, in order to compare the performance and efficiency of these similarity match algorithms.

In Section 3.3.1, a three-stage filtering is proposed to compare composite objects. To illustrate the idea, stage I is implemented in the function *equal()* in file composite.C. The function can be expanded by adding the code for stages II and III.

The main limitation on the similarity match of shape is the lack of an automatic shape extraction algorithm. In the absence of such an algorithm, the shape of an object has to be extracted manually from the image. The work is tedious and time-consuming, and the result is often subjective. For example, in Figure 5.3, the shape of the starfish was described as a circle instead of the star shape or polygon shape.

Figure 5.3: The shape of starfish

The graphical interface also imposes a limitation on color matching. Since a color is defined by three values, each of which can be any number from 0 to 255, the number of possible colors in the spectrum adds up to 256^3. Only a small selection of these (the most commonly used colors) are displayed by the interface for the user to choose. As a result, it is not surprising that the "no image is found to match this query" is returned to the user. To overcome this problem, the similarity threshold specified in the query has to be reduced so that more colors or images satisfy the query.

Chapter 6

Conclusion

6.1 Contributions

The research on image databases and image processing is complementary. In the image processing area, there is still a lot to be done before the feature extraction techniques and similarity match algorithms come to maturity. Given the methods and resources available, this report has achieved the objective of designing and implementing the framework of a content-based generic type system to support image storage and retrieval, as well as a query parser and engine to execute MOQL queries and perform similarity match. The main contributions of this report are as follows:

- The implementation proves that it is possible to extend the expressive power of OQL to arrive at a multimedia language, MOQL, which is capable of handling image queries on color, shape, texture and spatial relationship.
- In contrast to other designs, which are application specific, the DISIMA framework illustrates that a content-based generic type system can be implemented for image applications. The framework can be customized to meet specific needs of applications. An application can define its own LSO hierarchy, and the Schema Generator, included in the framework, will automatically generate the necessary code for compilation based on the user-defined LSO classes.
- The implementation of the DISIMA parser and engine provides the basic step on which the development of query indexing and optimization can be based.
- The DISIMA kernel supports objects of composite shapes. While other models focus on contours within contours, this implementation extends the definition of composite shape to include contours not inside each other.
- The DISIMA kernel implements the idea that allows the shape of an object to vary–an object can adopt different shapes at different zoom levels.
- The design of the resultset objects, together with the application of the conjunctive normal form concept, provides an efficient structure to store the information necessary for the retrieval of the final images (see Section 4.3.2).

6.2 Future Work

The research on image databases is an enormous topic and what is described in this report is only a small part of it. There is still a lot to be done, and this section will discuss some enhancements that can be considered in future development.

6.2.1 A full OQL parser and query processor

This report focuses on spatial relationship, color, shape, and texture–which are defined in the MOQL extension. In order to support the full MOQL, the current implementation needs to be integrated with an OQL parser and query processor. Instead of developing these from scratch, an alternative is to integrate with one already available. During query execution, the query is separated into two subqueries. One subquery is handled by the OQL parser/processor, and the other is handled by the MOQL extension parser/processor. The two intermediate results can then be consolidated.

6.2.2 Store images in progressive resolutions

The image resolution requirements vary in different applications. The current implementation stores the original image and its thumbnail. It is expected that other resolutions will also be stored so that the viewer can have a wider choice. An image can be displayed from low resolution to high resolution, allowing the viewer to exit at any point or go further to obtain more precise detail of the image.

6.2.3 Optimization

Some image retrieval systems apply real-time processing to analyze and compare images. DISIMA reduces the on-line response time by pre-processing. Image data representing color, shape, etc. are stored in the database. Analyzing the extracted data is faster than analyzing the raw image itself. The system can be further optimized and some suggestions are given below.

Searching the intermediate result instead of the database

When processing the first condition in a query, the search is always in the database. For example, the query

```
SELECT m
FROM image m, person p
WHERE m contains p;
```

searches the person_extent in the database when executing the condition

```
m contains p.
```

However, subsequent conditions can be processed based on the previous result. For example, the query

```
SELECT m
FROM image m, fish o
WHERE m contains o
AND o.color similar colorgroup(255,0,0);
```

has two conditions. When processing the second condition, the current query engine searches the colorgroup_extent and retrieves those matching colorgroup(255,0,0). The results of the two conditions are then consolidated with the *and* operator.

Since the result of the first condition reduces the search space to the set of fish objects, the query engine should have the option to search the fish_extent, instead of the colorgroup_extent, when executing the second condition. It will be faster because the number of colorgroup objects in the database is far greater than the number of fish objects. However, this option should be implemented together with some cost data and functions, in order to evaluate which option is more cost-efficient.

Indexing

Another way to optimize query processing is to use indices for similarity match. In the previous example, instead of scanning each object in the colorgroup_extent, an index based on binary tree [10], R-tree [25], or other algorithms (e.g. [21], [24]) can be implemented. For color matching, a 3-dimensional index is required to match the red, green, and blue values or the hue, saturation, and intensity values. Search by index is faster than a recursive search using brute force.

Manipulating the query tree

A query can be represented by different tree structures. All the six structures shown in Figure 6.1 produce the same query result. However, these tree structures require different processing speeds. The

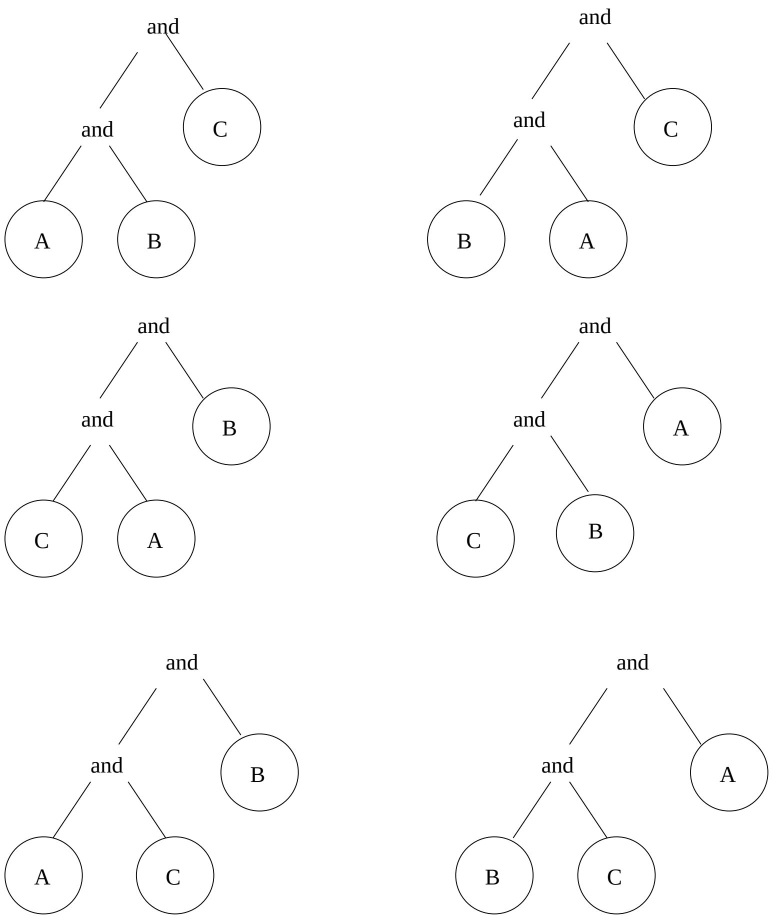


Figure 6.1: An example of a query represented by different tree structures

goal of a query optimizer is to select the structure which requires the least processing speed. In the current implementation, the leaves and nodes are built in the same order as the conditions are submitted in the input query string. For example, if the query is:

```
SELECT m
FROM image m
WHERE condition-A
AND condition-B
AND condition-C;
```

the tree structure created is like the top left one in Figure 6.1. To identify which structure is most efficient to execute requires implementing some cost functions.

6.2.4 Matching polygons with circles and ellipses

The turning angle algorithm applies to polygons, but not to shapes with curves, such as circles and ellipses. To solve the problem illustrated in Figure 3.29, either the turning angle algorithm has to be modified, or additional code is required to perform additional checking.

6.2.5 Query on the image hierarchy and image attributes

In the current implementation, only a LSO hierarchy, but not an image hierarchy, is defined. An image hierarchy can be added to the type system so that a query can specify the type of images to be retrieved, e.g., medical_image instead of all images.

6.2.6 Information about the image

When the final images are displayed, the user may want to obtain the text description of the images. One possibility is to allow the user to click on any image to obtain the text information.

6.2.7 Database update

In addition to storage and retrieval, the utimate goal is to allow image and object data to be updated in the database. The Vector class is defined for this purpose. Function calls can be initiated from the graphical interface. By clicking the translate, rotate, or scale button on the interface and supplying the necessary information, i.e., the angle of rotation, objects in the images can be manipulated and the modified objects can be stored in the database.

6.2.8 Replace the grade with a structure

The distance function assigns a grade to an object. If the final grade is greater than the similarity threshold specified in the query, the image containing that object will be retrieved by the engine. When more than one grade is involved in a query, the average is taken. To make the system more flexible, a structure, instead of a grade, can be returned by the distance functions. This structure contains the necessary information to compute the final grade. After all the conditions are executed, the user can then decide which formula (linear average, weighted average, Euclidean product, etc.) should be applied.

6.2.9 Query by sketch or example

Instead of describing the image features explicitly, a sample image or sketch can be used as the target object. The content of the target image is then extracted automatically and passed to the query engine for execution. This development requires advanced image processing techniques.

6.2.10 Extend to 3D still image

Instead of storing the x and y coordinates of a point, a 3-dimensional representation, i.e., x, y and z, can be stored in the type system. A 3D subclass can also be added to the Atomic class.

6.2.11 The video extension

The video feature can be incorporated by defining more data elements, such as the temporal element. The current implementation supports still images; a video image can be handled as a sequence of still images.

Bibliography

[1] http://whitechapel.media.mit.edu:80/vismod/demos/photobook/index.html.

[2] http://www.cs.ualberta.ca/~database/IDB/Interface.html.

[3] http://www.QBIC.almaden.ibm.com/stage/features.html.

[4] J Blakeley, W McKenna, and G Graefe. Experiences Building the Open OODB Query Optimizer. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 287–296, 1993.

[5] R. Cattell. *The Object Database Standard: ODMG-93 (Release 1.1).* Morgan Kaufmann, San Francisco, CA, 1994.

[6] S.K. Chang, Q.Y. Shi, and C.W. Yan. Iconic indexing by 2-D Strings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(3):413–427, May 1987.

[7] S. Chaudhuri and L. Gravano. Optimizing Queries over Multimedia Repositories. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 91–102, May 1996.

[8] W.W. Chu, I.T. Ieong, and R.K. Taira. A Semantic Modeling Approach for Image Retrieval by Content. *VLDB Journal*, 3:445–477, 1994.

[9] G. Cortelazzo, G.A. Mian, G. Vezzi, and P. Zamperoni. Trademark Shapes Description by String-Matching Techniques. *Pattern Recognition*, 27(8):1005–1018, 1994.

[10] M. Freeston. A General Solution of the n-dimensional B-tree Problem. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 80–91, 1995.

[11] R.C. Gonzalex and R.E. Woods. *Digital Image Processing.* Addison-Wesley, New York, 1993.

[12] V.N. Gudivada and G.S. Jung. An Algorithm for Content-based Retrieval in Multimedia Databases. *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 56–61, June 1996.

[13] V.N. Gudivada and V.V. Raghavan. Design and Evaluation of Algorithms for Image Retrieval by Spatial Similarity. *ACM Transaction on Information Systems*, 13(2):115–144, April 1995.

[14] D. Hearn and M.P. Baker. *Computer Graphics 2nd edition.* Prentice Hall International Editions, New York, 1984.

[15] IBM. Query by Image and Video Content: The QBIC System. *IEEE COMPUTER Innovative technology for computer professionals*, pages 23–32, september 1995.

[16] S. Kelly and F. Vincent. Nearest Neighbor Queries. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[17] J.R. Levine, T. Mason, and D. Brown. *Unix Programming Tools: Lex and Yacc.* O'Reilly and Associates, Inc., California, 1995.

[18] J.Z. Li, M.T. Özsu, D. Szafron, and V. Oria. MOQL: A Multimedia Object Query Language. *3rd International Workshop on Multimedia Information Systems, Como, Italy*, pages 19–28, September 1997.

[19] F. Liu and R.W. Picard. Periodicity, Directionality, and Randomness: Wold Features for Image Modelling and Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(7):722–733, July 1996.

[20] R.C. Nelson and H. Samet. A Consistent Hierarchical Representation for Vector Data. *ACM SIGGRAPH journal*, 20(4):197–206, August 1986.

[21] Y. Niu, M.T. Özsu, and X. Li. 2D-h Trees: An Index Scheme for Content-Based Retrieval of Images in Multimedia Systems. *Proceedings of IEEE International Conference on Intelligent Processing Systems, ICIPS*, pages 1710–1715, October 1997.

[22] V. Oria, M.T. Özsu, L. Liu, X. Li, J.Z. Li, Y. Niu, and P.J. Iglinski. Modeling images for content-based queries: The DISIMA approach. *2nd International Conference on Visual Information Systems, San Diego, CA*, pages 339–346, December 1997.

[23] V. Oria, M.T. Özsu, B. Xu, L. I. Cheng, and P.J. Iglinsk. VisualMOQL: The DISIMA Visual Query Language. *Department of Computing Science, University of Alberta.*

[24] D. Papadias, Y. Theodoridis, T. Sellis, and M.J. Egenhofer. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 92–103, 1995.

[25] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^{+}-tree: A Dynamic Index for multi-dimensional Objects. *Proceedings of the 13th International Conference on Very Large Databases, VLDB*, pages 507–517, 1987.

[26] A. Soffer and H. Samet. Pictorial Queries by Image Similarity. *Proceedings of IEEE International Conference on Pattern Recognition*, pages 114–119, 1996.

[27] M. Stricker and M. Orengo. Similarity of Color Images. *Proceedings of Storage and Retrieval for Images and Video Databases II, IS&T/SPIE Symposium on Electronic Imaging Science and Technology*, February 1995.

[28] A. Vailaya, Y. Zhong, and A.K. Jain. A Hierarchical System for Efficient Image Retrieval. *Proceedings of IEEE International Conference on Pattern Recognition*, pages 356–359, 1996.

[29] B. Xu. A Visual Query Facility for Image Databases. *M.Sc. Thesis, University of Alberta.*

Appendix A

An extraction of the code on I_class and C_class

```
class I_Polygon {
protected:
  os_List<Point*>* head;
};

class I_Rectangle {
protected:
  Point center;
  Point corner1; // corner1&2 are consecutive and in clockwise direction
  Point corner2;
};

class I_Square {
protected:
  Point center;
  Point corner;
};

class C_Polygon : protected I_Polygon, public Polygon {
private:
public:
  C_Polygon(os_List<Point*> *h);
  ~C_Polygon();
  void set_head(os_List<Point*>* p);
  os_List<Point*>* get_head();
};

class C_Rectangle : protected I_Rectangle, public Rectangle {
private:
public:
  C_Rectangle();
  C_Rectangle(Point& p0, Point& p1, Point& p2);
  ~C_Rectangle();
```

```
  os_List<Point*>* get_head();
  Point* get_center();
  Point* get_corner1();
  Point* get_corner2();
  void set_center(Point* p);
  void set_corner1(Point* p);
  void set_corner2(Point* p);
};

class C_Square : protected I_Square, public Square {
private:
public:
  C_Square(Point& p1, Point& p2);
  ~C_Square();
  os_List<Point*>* get_head();
  Point* get_center();
  Point* get_corner();
  void set_center(Point* p);
  void set_corner(Point* p);
};

class Polygon : public TwoD {
private:
public:
  static C_Polygon* Create(os_List<Point*>* h);
  ~Polygon();
  virtual os_List<Point*>* get_head()=0;
  Point* get_center();
  int get_count();
  void print();
  virtual void* return_this() { return this; }
  Boolean equal(Polygon* sh);
};

class Rectangle : public Polygon {
private:
public:
  static C_Rectangle* Create(Point* c, Point* c1, Point* c2);
  ~Rectangle();
  virtual os_List<Point*>* get_head()=0;
  virtual void* return_this() { return this; }
};

class Square : public Rectangle {
private:
public:
  static C_Square* Create(Point* c, Point* c1);
  ~Square();
  virtual os_List<Point*>* get_head()=0;
```

```
  virtual void* return_this() { return this; }
};
```

Appendix B

An extraction of the parser rules

A new predicate, called image_comparison_predicate, is added to the parser rules.

```
image_comparison_predicate:
                contain_predicate
        |       shape_comparison_predicate
        |       color_comparison_predicate
        |       mbb_comparison_predicate
        |       texture_comparison_predicate
        ;
contain_predicate:
                NAME CONTAINS NAME
        ;
shape_comparison_predicate:
                shape_variable SIMILAR shape_right_exp
        ;
color_comparison_predicate:
                color_variable SIMILAR color_right_exp
        ;
mbb_comparison_predicate:
                mbb_variable SPATIALFUNC mbb_right_exp
        ;
texture_comparison_predicate:
                texture_variable SIMILAR texture_right_exp
        ;
shape_right_exp:
                NAME '.' SHAPETYPE
        |       composite_shape_function
        |       shape_function
        |       '(' shape_right_exp ')'
        ;
color_right_exp:
                NAME '.' COLORTYPE
        |       color_function
        |       '(' color_right_exp ')'
        ;
mbb_right_exp:
```

```
                NAME '.' MBBTYPE
        |       '(' mbb_right_exp ')'
        ;
texture_right_exp:
                NAME '.' TEXTURETYPE
        |       texture_function
        |       '(' texture_right_exp ')'
        ;
shape_variable:
                NAME '.' SHAPETYPE
        ;
color_variable:
                NAME '.' COLORTYPE
        ;
mbb_variable:
                NAME '.' MBBTYPE
        ;
texture_variable:
                NAME '.' TEXTURETYPE
        ;
composite_shape_function:
                COMPOSITEFUNC '(' shape_list ')'
        |       COMPOSITEFUNC
        ;
shape_list:
                shape_function
        |       shape_list shape_function
        ;
shape_function:
                POINTFUNC '(' point_spec ')'
        |       POINTFUNC
        |       POLYGONFUNC '(' point_spec point_spec point_spec point_list ')'
        |       POLYGONFUNC
        |       POLYLINEFUNC '(' point_spec point_spec point_list ')'
        |       POLYLINEFUNC
        |       SEGMENTFUNC '(' point_spec point_spec ')'
        |       SEGMENTFUNC
        |       CIRCLEFUNC '(' point_spec point_spec ')'
        |       CIRCLEFUNC
        |       SQUAREFUNC '(' point_spec point_spec ')'
        |       SQUAREFUNC
        |       ELLIPSEFUNC '(' point_spec point_spec point_spec')'
        |       ELLIPSEFUNC
        |       RECTANGLEFUNC
        |       TRIANGLEFUNC '(' point_spec point_spec point_spec')'
        ;
shape_class:
                POINTFUNC
        |       POLYGONFUNC
```

```
        |       POLYLINEFUNC
        |       SEGMENTFUNC
        |       CIRCLEFUNC
        |       SQUAREFUNC
        |       ELLIPSEFUNC
        |       RECTANGLEFUNC
        |       TRIANGLEFUNC
        |       COMPOSITEFUNC
        ;
point_list:
                point_spec
        |       point_list point_spec
        ;
point_spec:
                INTNUM ',' INTNUM
        ;
color_function:
                COLORFUNC '(' color_list ')'
        ;
color_list:
                color_spec
        |       color_list color_spec
        ;
color_spec:
                INTNUM ',' INTNUM ',' INTNUM
        ;
texture_function:
                TEXTUREFUNC '(' texture_list ')'
        ;
texture_list:
                texture_spec
        |       texture_list texture_spec
        ;
texture_spec:
                INTNUM
        ;
```

Appendix C

An example of the user defined file *userClasses*

```
####################################################################
#
# DON'T DELETE THE FOLLOWING INSTRUCTION
#
# file: userClasses
#
# Description: This file is defined by the user but has to follow
# the format below.
#
# Format: Reserved words must be in lower case.
#   Reserved words- class, public, char*, int, float
#   There is no space before class, // and } and no space
#   after } and ;.
#   Single comment can be preceded by //
#
# Note: The superclass has to be defined before its subclass
#
####################################################################

// comment
class Person  : public Lso {
  char* lastName;
  char* firstName;
  char* middleName;
  char* nationality;
  char* sex;
  int yearOfBirth;
};

class Athlete  : public Person {
// comment
  char* sport;
  char* team;
```

```
};

class Politician  : public Person {
  char* party;
  char* officeHeld;
};

class MovieStar  : public Person {
  char* films;
};

class Scientist  : public Person {
  char* field;
};

class Animal  : public Lso {
  char* comName;
  char* sciName;
};

class Mammal  : public Animal {
};

class Bird  : public Animal {
};

class Reptile  : public Animal {
};

class Insect  : public Animal {
};

class Crustacean  : public Animal {

};

class Fish  : public Animal {
};

class OtherAnimal  : public Animal {
};

class Vehicle  : public Lso {
  char* type;
};

class Plane  : public Vehicle {
};
```

```
class Ship  : public Vehicle {
};

class Car  : public Vehicle {
  char* make;
  char* model;
  int year;
};

class Truck  : public Vehicle {
  char* make;
};

class Motorcycle  : public Vehicle {
  char* make;
};

class Bicycle  : public Vehicle {
};

class Building  : public Lso {
  char* city;
  char* prov_state;
  char* country;
};

class Hospital  : public Building {
  char* name;
};

class School  : public Building {
  char* name;
};

class Stadium  : public Building {
  char* name;
};

class House  : public Building {
  char* owner;
  char* resident;
  char* address;
};

class Apartment  : public Building {
  char* name;
  char* address;
};
```

```
class Office  : public Building {
  char* name;
};

class Museum  : public Building {
  char* name;
};
```

Appendix D

Glossary

DBMS Database Management System

DISIMA The Distributed Image Database Management System

GIS Geographic Information System

GUI Graphical User Interface

HTML Hyper Text Markup Language

LSO Logical Salient Object

MOQL Multimedia Object Query Language

Mbb Minimum bounding box

OQL Object Query Language

PSO Physical Salient Object

SQL Structural Query Language

associative access –Find objects with certain properties in the database

content-based –Based on the image content, that is color, shape, texture, and spatial features.

distance function –An equation used to compute the similarity between the database object and the target object. A grade between 0 and 1 is returned after the computation. An exact match is represented by 1.

extent –A collection of objects with similar properties.

impedance mismatch –Problem between a database system and its programming language, where the structures provided by the database system are distinct from those provided by the programming language.

navigational access –Further investigate the object based on any embedded oids after the object is found.

oid –Object identity or pointer.

resultpso object –Each resultset object defines a set of resultpso objects. A resultpso object is associated with an object label and a set of unitpso objects.

resultset object –The intermediate result of a query is stored as a set of resultset objects.

unitpso object –Each unitpso object is associated with a PSO pointer and the similarity grade assigned to that PSO object.