Contents

Chapter 0: Acknowledgement & team work.	2
Chapter 1: introduction:	3
✓ Problem statement.	4
✓ Image processing.	5
✓ Image retrieval.	9
✓ System analysis & design:	11
• System analysis.	11
• System design .	18
Chapter 2: algorithms:	24
✓ Comparison algorithms.	31
✓ Filtering algorithms.	34
Chapter 3: implementation:	51
✓ Interface implementation codes.	52
✓ Comparison implementation code.	54
✓ Filter implementation codes.	55
Chapter 4: conclusion & future work:	58

Chapter 0

Acknowledgement & Team work

Acknowledgement:

We would like to thank Dr.ibraheem AL_hennawy or his guidance and support throught the duration of the project .We also want to thank the committee members Dr.wael said and Dr.Ibraheem AL_hennawy again,for their support and assistance throughout the entire process . We especially want to thank Eng. Mahmoud Mahdi the assistant professor for his valuable experience and input , which contributed greatly to the project .

We would also like to thank the friends who were there for me throughout this process . Without your help , we wouldn't have made it this far!

Team work

Team work:

1-Abdullah Ibraheem AL_Nahhal.	CS
2- Mhamed bushra	CS
3- Ahmed Abd EL_Azeem AL_Azab.	CS
4- Mohamed Salah	IT
5-Remon Alfonse	IT

CHAPTER I

Introduction

1.1: Problem statement:

Mr.X goes to the airport and on the check bank of the passport the police officer stopped him because his name was the same name of a wanted person .

Mr.X waited until they have been sure from him that he is a good person.

But when they let him go the airplane was gone and it lifted him so he suggested that if there was a program that confirms people by there images (faces) it will be so fast to avoid that confusion .

From that moment we took on our hands to establish that program for our countries rise and to be proud of us its sons .

1.2: Image processing:

1.2.1 Images and pictures

human beings are predominantly visual creatures: we rely heavily on our vision to make sense of the world around us.

We not only lo ok at things to identify and classify them, but we can scan for differentiate , and obtain an overall rough feelingfor a scene with a quick glance.

Humans have evolved very precise visual skills: we can identify a face in an instant; we can differentiate colors; we can process a large amount of visual information very quickly .

However, the world is in constant motion: stare at something for long enough and it will change in some way .

Even a large solid structure, like a building or a mountain, will change its appearance depending on the time of day (day or night); amount of sunlight (clear or cloudy), or various shadows falling upon it.

We are concerned with single images: snapshots, if you like, of a visual scene. Although image processing can deal with changing scenes, we shall not discuss it in any detail in this text.

For our purposes, an image is a single picture which represents something.

It may be a picture of a person, of people or animals, or of an outdoor scene, or a microphotograph of an electronic component, or the result of medical imaging. Even if the picture is not immediately recognizable, it will not be just a random blur.

1.2.2 What is image processing?

Image processing involves changing the nature of an image in order to either

1. improve its pictorial information for human interpretation,

2. render it more suitable for autonomous machine perception.

We shall be concerned with digital image processing, which involves using a computer to change the nature of a digital image (see below). It is necessary to realize that these two aspects represent two separate but equally important aspects of image processing. A procedure which satisfies condition (1)a procedure which makes an image look bettermay be the very worst procedure for satisfying condition (2).

Humans like their images to be sharp, clear and detailed; machines prefer their images to be simple and uncluttered.

Examples of (1) may include:

- Enhancing the edges of an image to make it appear sharper; an example is shown in figure 1.1.

  Note how the second image appears cleaner; it is a more pleasant image. Sharpening edges is a vital component of printing: in order for an image to appear at its beston the printed page; some sharpening is usually performed.

(a) The original image (b) Result after "sharperning"

Figure 1.1: Image sharperning

- Removing noise from an image; noise being random errors in the image. An example is given in figure 1.2. Noise is a very common problem in data transmission: all sorts of electronic components may affect data passing through them, and the results may be undesirable. As we shall see in chapter 5 noise may take many different forms ;each type of noise requiring a different method of removal.
- Removing motion blur from an image. An example is given in figure 1.3. Note that in the deblurred image (b) it is easier to read the numberplate, and to see the spikes on the fence behind the car, as well as other details not at all clear in the original image (a). Motion blur may occur when the shutter speed of the camera is too long for the speed of the object. In photographs of fast moving objects: athletes, vehicles for example, the problem of blur may be considerable.

Examples of (2) may include:

- Obtaining the edges of an image. This may be necessary for the measurement of objects in an image; an example is shown in figures 1.4. Once we have the edges we can measure their spread, and the area contained within them. We

can also use edge detection algorithms as a first step in edge enhancement, as we saw above.

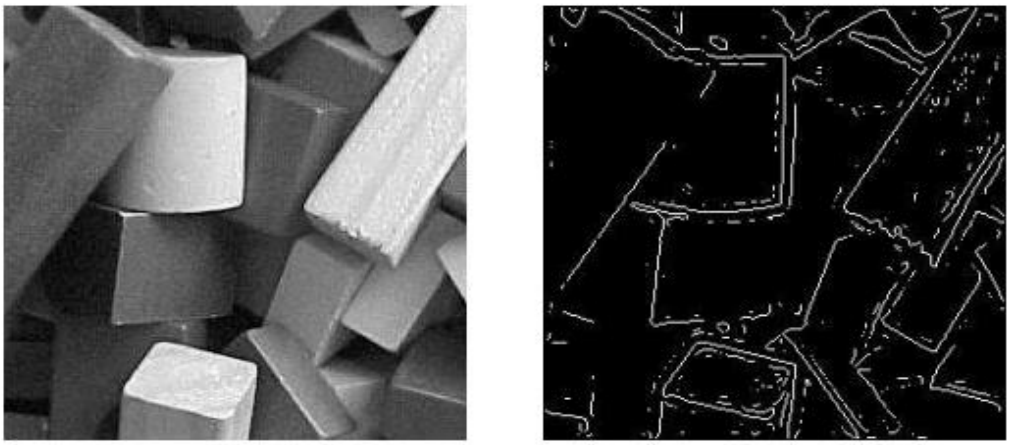
(a) The original image (b) After removing noise

Figure 1.2: Removing noise from an image

(a) The original image (b) After removing the blur

Figure 1.3: Image deblurring

From the edge result, we see that it may be necessary to enhance the original image slightly ,to make the edges clearer.



(a) The original image (b) Its edge image

Figure 1.4: Finding edges in an image

- Removing detail from an image. F or measurement or counting purposes, we may not be interested in all the detail in an image. For example, a machine inspected items on an assembly line, the only matters of interest may be shape, size or colour. F or such cases, we might want to simplify the image. Figure 1.5 shows an example: in image (a) is a picture of an Africa buffalo, and image (b) shows a blurred version in which extraneous detail (like the logs of wood in the background) have been removed. Notice that in image (b) all the ne detail is gone; what remains is the coarse structure of the image. We could for example, measure the size and shape of the animal without being distracted by unnecessary detail.

(a) The original image (b) Blurring to remove detail

Figure 1.5: Blurring an image

1.3 image retrieval:

An ***image retrieval*** system is a computer system for browsing, searching and retrieving images from a large database of digital images. Most traditional and common methods of image retrieval utilize some method of adding metadata such as captioning, keywords, or descriptions to the images so that retrieval can be performed over the annotation words. Manual image annotation is time-consuming, laborious and expensive; to address this, there has been a large amount of research done on automatic image annotation. Additionally, the increase in social web applications and the semantic web have inspired the development of several web-based image annotation tools.

The first microcomputer-based image database retrieval system was developed at MIT, in the 1990s, by Banireddy Prasaad, Amar Gupta, Hoo-min Toong, and Stuart Madnick.[1]

A 2008 survey article documented progresses after 2007

A ***database*** is an organized collection of data. The data is typically organized to model relevant aspects of reality (for example, the availability of rooms in hotels), in a way that supports processes requiring this information (for example, finding a hotel with vacancies).

A ***digital image*** is a numeric representation (normally binary) of a two-dimensional image. Depending on whether the image resolution is fixed, it may be of vector or raster type. Without qualifications, the term "digital image" usually refers to raster images also called bitmap images.

The term ***metadata*** refers to "data about data". The term is ambiguous, as it is used for two fundamentally different concepts (types). ***Structural metadata*** is about the design and specification of data structures and is more properly called "data about the containers of data"; ***descriptive metadata***, on the other hand, is about individual instances of application data, the data content. In this case, a useful description would be "data about data content" or "content about content" thus meta content. Descriptive, Guide and the National Information Standards Organization concept of administrative metadata are all subtypes of meta content.

Automatic image annotation (also known as automatic image tagging or linguistic indexing) is the process by which a computer system automatically assigns metadata in the form of captioning or keywords to a digital image.

1.3.1 search methods:

Image search is a specialized data search used to find images. To search for images, a user may provide query terms such as keyword, image file/link, or click on

some image, and the system will return images "similar" to the query. The similarity used for search criteria could be meta tags, color distribution in images, region/shape attributes, etc.

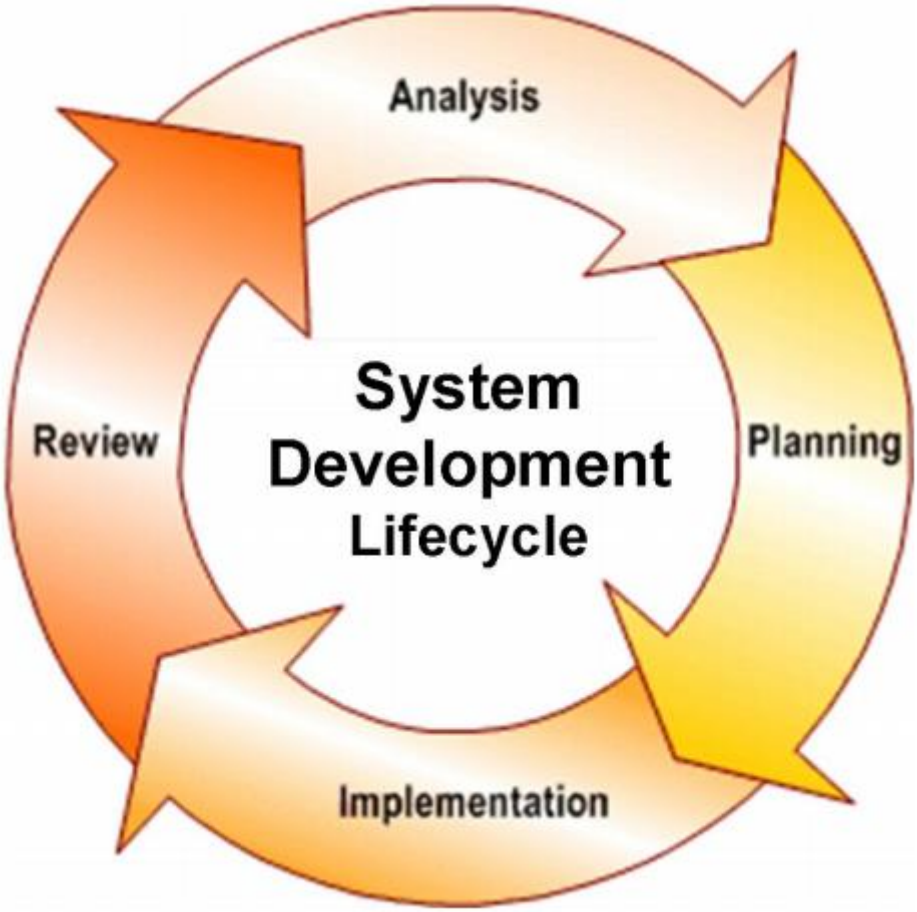
- ***Image meta search*** - search of images based on associated metadata such as keywords, text, etc.
- ***Content-based image retrieval*** (CBIR) – the application of computer vision to the image retrieval. CBIR aims at avoiding the use of textual descriptions and instead retrieves images based on similarities in their contents (textures, colors, shapes etc.) to a user-supplied query image or user-specified image features.
- ***List of CBIR Engines*** - list of engines which search for images based image visual content such as color, texture, shape/object, etc.

1.3.2 Data scope:

It is crucial to understand the scope and nature of image data in order to determine the complexity of image search system design. The design is also largely influenced by factors such as the diversity of user-base and expected user traffic for a search system. Along this dimension, search data can be classified into the following categories:

- *Archives* - usually contain large volumes of structured or semi-structured homogeneous data pertaining to specific topics.
- *Domain-Specific Collection* - this is a homogeneous collection providing access to controlled users with very specific objectives. Examples of such a collection are biomedical and satellite image databases.
- *Enterprise Collection* - a heterogeneous collection of images that is accessible to users within an organization's intranet. Pictures may be stored in many different locations.
- *Personal Collection* - usually consists of a largely homogeneous collection and is generally small in size, accessible primarily to its owner, and usually stored on a local storage media.
- *Web* - World Wide Web images are accessible to everyone with an Internet connection. These image collections are semi-structured, non-homogeneous and massive in volume, and are usually stored in large disk arrays.

1.4 System Analysis & design:



1.4.1 Systems Analysis Definition – (SAD):

The analysis of the role of a proposed system and the identification of the requirements that it should meet. SAD is the starting point for system design. The term is most commonly used in the context of commercial programming, where software developers are often classed as either systems analysts or programmers.

The systems analysts are responsible for identifying requirements (i.e. systems analysis) and producing a design. The programmers are then responsible for implementing it.

1.4.2 System Analysis:

In this phase, the current system is studied in detail. A person responsible for the analysis of the system is known as analyst. In system analysis, the analyst conducts the following activities.

1.4.3 Needs Analysis:

This activity is known as requirements analysis. In this step the analyst sums up the

requirements of the system from the user and the managers. The developed system should satisfy these requirements during testing phase.

1.4.4 Data Gathering:

In this step, the system analyst collects data about the system to be developed. He uses different tools and methods, depending on situation.

1.4.5 Written Documents:

The analyst may collect the information/data from written documents available from manual-files of an organization. This method of data gathering is normally used if you want to computerize the existing manual system or upgrade the existing computer based system. The written documents may be reports, forms, memos, business plans, policy statements, organizational charts and many others. The written documents provide valuable information about the existing system.

1.5 Interviews:

Interview is another data gathering technique. The analyst (or project team members) interviews, managers, users/ clients, suppliers, and competitors to collect the information about the system. It must be noted that the questions to be asked from them should be precise, relevant and to the point.

Advantages of using an Interview:

- If the respondent lacks reading skills to answer a questionnaire.
- Are useful for untangling complex topics.
- The Interviewer can probe deeper into a response given by an interviewee.
- Interviews produce a higher response rate.

Disadvantages of using an Interview:

- the interviewer can affect the data if he/she is not consistent.
- It is very time consuming.
- It is not used for a large number of people.
- The Interviewer may be biased and ask closed questions .

1.5.1 Questionnaires :

Questionnaires are the feedback forms used to collect Information. The interview technique to collect information is time -consuming method, so Questionnaires Are designed to collect information from as many people as we like. It is very convenient and inexpensive method to collect information but sometimes the response may be Confusing or unclear and insufficient.

Advantages of using Questionnaires :

Scanning can be the fastest method of data entry for paper questionnaires.

Scanning is more accurate than a person in reading a properly completed questionnaire.

Disadvantages of using Questionnaires:

Scanning is best-suited to "check the box" type surveys and bar codes. Scanning programs have various methods to deal with text responses, but all require additional data entry time.

Scanning is less forgiving (accurate) than a person in reading a poorly marked questionnaire.

1.5.2 Observations:

In addition to the above -mentioned three techniques to collect information, the analyst (or his team) may collect Information through observation. In this collect technique, the working, behavior, and other related information of the existing system are observed. It means that working of existing system is watched carefully.

Advantages of Using Observations :

- You get to know the child well.
- It enables you to gain an insight into the uniqueness of the child.
- It allows you to obtain a better understanding of the „norms „of development.
- It enables you to chart development changes over a period of time.
- As the study is over a period of time you may uncover an area of concern, this may enable you to ensure help/guidance is offered earlier than otherwise have been.

Disadvantages of Using Observations:

• The child may be absent from the setting for a long period of time or leave the setting. (It is therefore recommended that you start the study with two children.)

• Relationships with parents may become strained due to the continuous observation of the child.

• Objective observations may upset parent/ career.

• If a child's behavior or development proves to be atypical (not typical) this may give a distorted view of normal behavior and developmental norms.

• Issues around confidentiality may be raised, as it may be easy for others to identify the child.

1.5.3 Sampling:

If there are large numbers of people or events involved in the system, we can use sampling method to collect information. In this method, only a part of the people or events involved are used to collect information. For example to test the quality of a fruit, we test a piece of the fruit.

Advantages of Using Sampling:

• A collection of precise data/when completed data is readily accessible.

• It is quick and easy to use.

• It is more closely focused.

• It can reveal unsuspected patterns of behavior.

Disadvantages of Using Sampling:

• Allocating time to complete the task (may need to take place over a long

period of time).

• It needs to be carefully prepared.

• Being aware of the passage of time when doing time samples.

• Keeping one child insight at all times remembering to be unob trusive as

possible

1.5.4 Data Analysis:

After completion of *Data Gathering* step the collected data about the system is analyzed to ensure that the data is accurate and complete. For this purpose, various tools may be used. The most popular and commonly used tools for data analysis are:

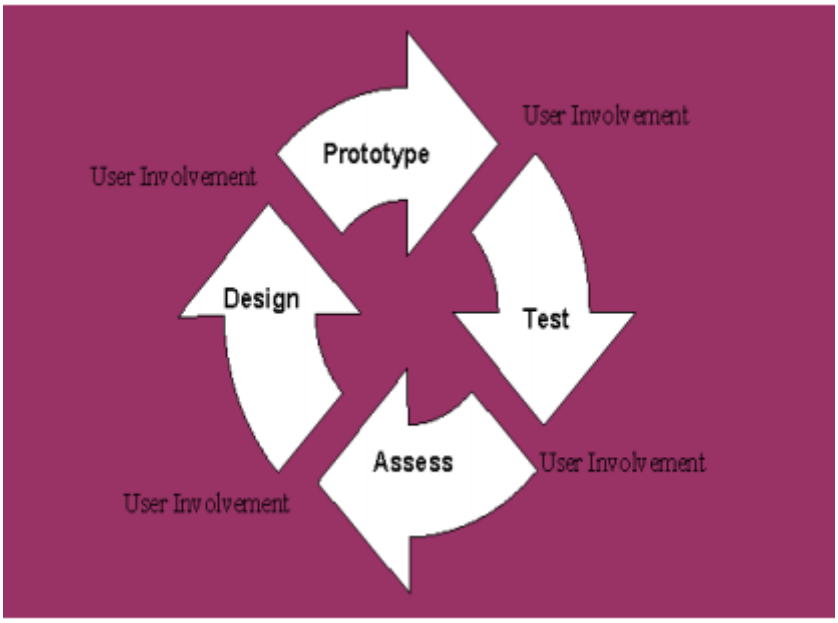
* DFDs (Data Flow Diagrams)
* System Flowcharts
* Connectivity Diagrams
* Grid Charts
* Decision Tables etc.

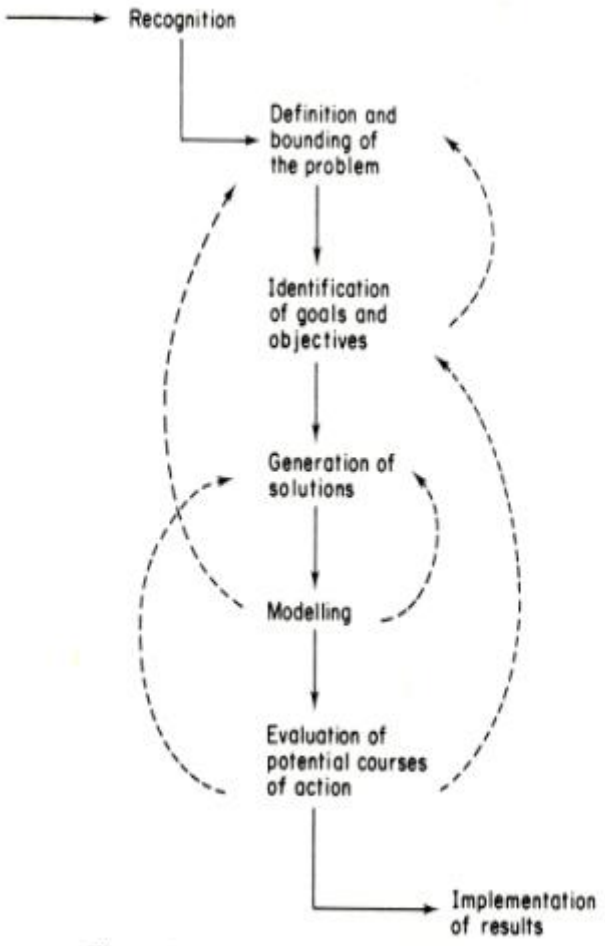
1.5.5Analysis Report:

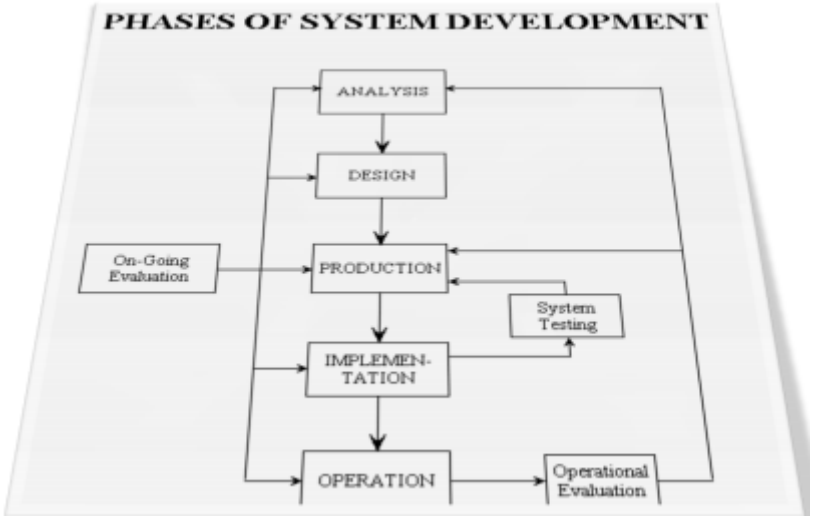
After completing the work of analysis, the requirements collected for the system are documented in a presentable form. It means that the analysis report is prepared.

It is done for review and approval of the project from the higher management. This report should have three parts.

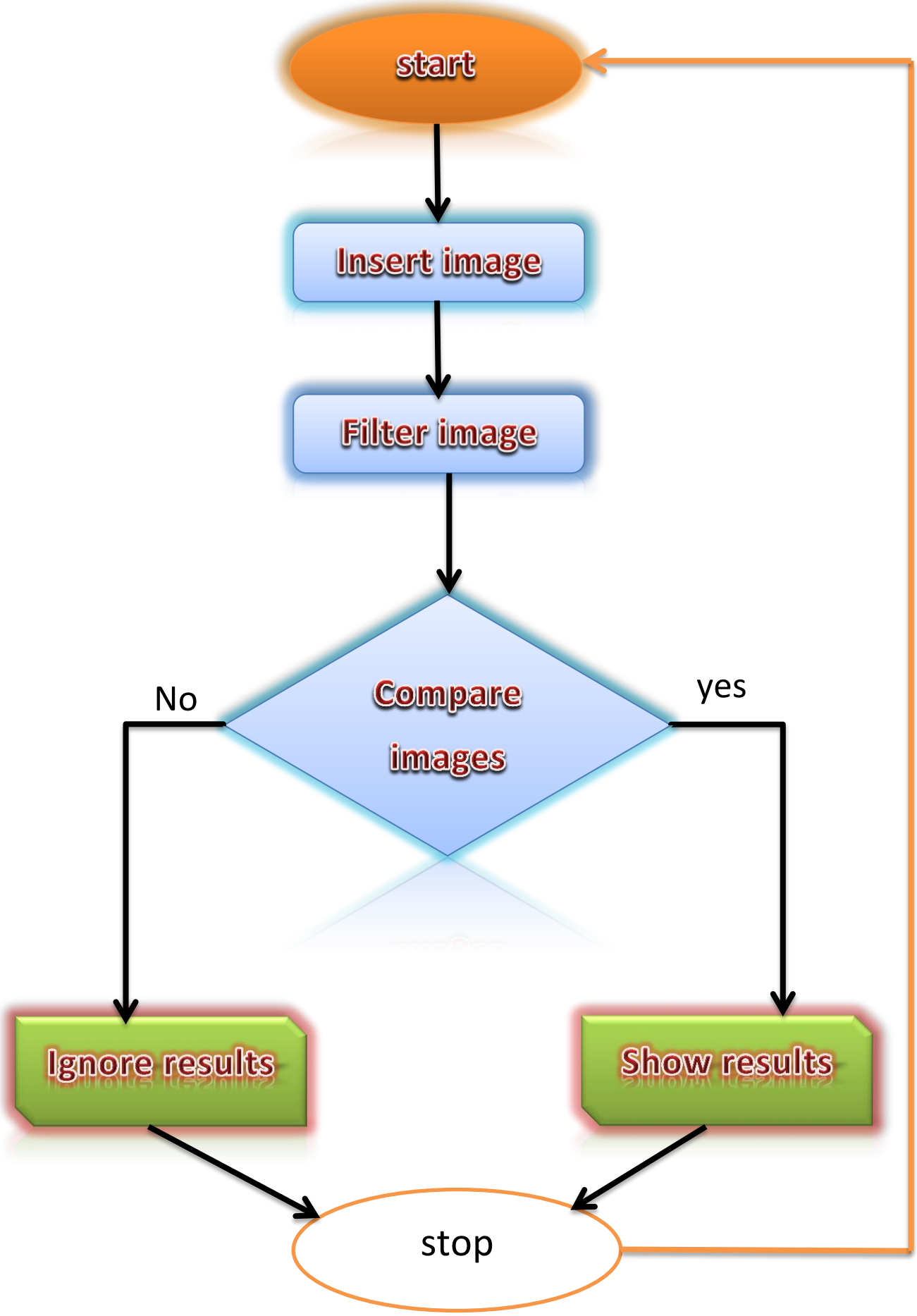
- First, it should explain how the current system works.
- Second, it should explain the problems in the existing system.
- Finally, it should describe the requirements for the new system and make recommendations for future.




Recognition
Definition and bounding of the problem
Identification of goals and objectives
Generation of solutions
Modelling
Evaluation of potential courses of action
Implementation of results


PHASES OF SYSTEM DEVELOPMENT
ANALYSIS
DESIGN
On-Going Evaluation
PRODUCTION
System Testing
IMPLEMEN-TATION
OPERATION
Operational Evaluation

1.6 Flow Chart Sequence:



1.7 Systems design:

Is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements?

Systems Design Techniques:

- Logical Data Modeling
- Entity/Event Modeling
- Logical Data Modeling

The objectives of this section are to provide definitions for the terms Logical Data Model (LDM) and Logical Data Structure (LDS), and to explain the constituent parts of a Logical Data Structure, and finally to introduce a step by step process for constructing Logical Data Models.

1.7.1 What is a Logical Data Model?

A Logical Data Model (LDM) is a representation of the data used by a system. It shows how the data is logically grouped and the relationships between these groupings as defined by the business requirements of the system.

The LDM comprises:-

A diagram called a Logical Data Structure (LDS). NB LDS is simply the SSADM terminology for a Data Model or Entity- Relationship Model).

Associated documentation of entities and relationships.

1.7.2 What do Logical Data Structures consists of?

Logical Data Structures consist of entity types and relationship types:-

An entity type is a logical grouping of data which is relevant to the application in question. The entity type must be relevant, e.g. an information system specifically for the Learning Resources Centre (LRC) would not hold information about lecturer's cars, since this is not relevant.

An entity type is an identifiable object, classification, concept, activity, event or thing concerning the application. The thing must be identifiable since if it cannot be identified no information can be recorded about it in a database, e.g. a chair within the library isn't identifiable so wouldn't be an entity, however the classification chair type possibly would be an entity.

An entity is an occurrence of an entity type. The terms entity type and entity are often used interchangeably, the context usually defines what is actually meant. There must be the possibility of an entity type having more than one occurrence. A common mistake is to include one "super‟ entity type in the LDS representing the company, the garage the library

or the surgery or whatever.

For example in a LDS for the University of Glam organ's LRC, LRC is not an entity type since there is only one occurrence. If however a LDS is being developed for all the college libraries in Wales then LRC is an entity type because there is the possibility of more than one occurrence.

1.8 Entity types :

Are represented as soft rectangles containing the name of the entity. Naming of entities is critical, especially when groups of people are working together. An agreed definition of what is actually meant by an entity name will avoid a lot of confusion and pointless discussion.

Relationships may be recursive, i.e. an entity can be related to other entities of the same type. Recursive relationships can be 1: M (e.g. a tree structure like an organization chart) or M: M (e.g. a bill of materials structure).

M: M recursive relationships are broken down into two 1: M relationships and a

link entity in the same way as non-recursive M: M relationships.

Relationships can be mutually exclusive with other relationships. In other words an

occurrence of entity type A may be owned by either an occurrence of entity type B or an occurrence of entity type C. Consider an application in which there is an entity type called Course Type, e.g. SSADM Version 4 and an entity type called Course Run, e.g. SSADM Version 4 at the Marriot Hotel Cardiff on July 19th 1994. If there are places available on a particular Course Run then a booking may be created owned by that Course Run. If there aren't any places available on any Course Runs, then a booking can be owned by the Course Type indicating that the booking is on a waiting list. When a place becomes available the booking can be disconnected from the Course Type and connected to the appropriate Course Run.

Alternatively an occurrence of entity type A can either own occurrences of entity

type B or occurrences of entity type C. Consider an application which needs to

trace who has bought particular parts and who as supplied particular parts. An Organization entity could either own many Supplier_Of_Part entities or many Purchaser_Of_Part entities. Mutual exclusion is shown using an arc across the relationship lines.

Each end of each relationship must be optional or mandatory. If a relationship end is optional (shown by using a broken line) the entity at that end of the relationship can exist without taking part in the relationship. If a relationship end is mandatory (shown using a solid line) the entity at that end of the relationship must take part in the relationship.

This gives rise to four types of one to much relationship:

- **Owner Optional - Member Mandatory.**

  Consider an application in which a Customer can own many Orders. A Customer entity is allowed to exist without having placed any Orders (e.g. a potential customer), but an order must have been placed buy a customer.

- **Owner Mandatory - Member Mandatory.**

  Consider an application in which an Order consists of many Order Lines. In this case an Order must have at least one Order Line (an order consisting of 0 order lines is nonsensical) and an Order Line must be owned by an Order.

- **Owner Optional - Member Optional**

  Consider an application in some Employees negotiate Orders with Suppliers, but Orders can be received direct from Customers. In this case an Employee (who isn't a salesperson) can exist without negotiating any Orders and an .Order isn't necessarily owned by an Employee.

- **Owner Mandatory - Member Optional.**

  Consider an application where some Employees are paid extras via a privately negotiated Commission Plan. In this case a Commission Plan would not exist unless there was at least one employee being paid via that Commission Plan. An Employee does not have to be on a Commission Plan.

Relationships are named at both ends, the names chosen should be such that

meaningful sentences can be constructed describing the nature of the relationship

using the entity names and the relationship names.

1.8.1 The Relationship between Logical Data Structures and Data Flow Diagrams.

Since LDSs and DFDs are different views of the same thing you would expect there to be some commonality between them. The obvious area is data stores and entities. Each entity type in the LDS has to be represented in a data store somewhere. This may be a one to one mapping, e.g. the customer entity type will map one to one on to the customer's data store, or a many to one mapping, e.g. the order and order line entity types will be held in one data store called orders .

1.8.2 How is Logical Data Structures Created?

The following steps may be helpful but there really are no hard and fast rules. As the analysis and design exercise proceeds the LDS will evolve and many re-drafts may be necessary as the analysts understanding of the application improves: -

- Identify an initial list of entities
- Using an Entity/Relationship cross reference identify the initial relationships
- Create a first draft LDS
- Validate the LDS against the identified requirements
- Identify any new entities/relationships required
- Rationalize the LDS by combining removing entities/relationships
- Re-Draft the LDS
- Identify and place the required attributes, ensuring that each entit y has the
- appropriate primary and foreign keys
- Ensure that the structure is in third normal

1.8.3 How are Logical Data Structures and Data Flow Diagrams related?

Each entity type in the LDS will have an associated ELH; each event has to be supported by a process or processes in the DFD. Entity/Event modeling in addition to providing a useful system viewpoint in their own right can be used to check the consistency, accuracy and completeness of the LDS and DFDs.

As can be seen analysis is not simply a case of drawing the LDS, drawing the

DFDs and then drawing the ELHs, frequently the analyst will have to change tack and move from LDS to DFD to ELH, modifying and re-drafting as understanding improves.

At the end of the analysis and logical design stages the analysis/design team should have three separate but linked models which have been cross validated and which together give a complete picture of the system questions.

1.9 Interface Problems:

Interface problem defined by many who interested in this field. One of those professional designer and researcher is Galatz .Galatz said these problems result in confusion, panic, frustration, boredom, misuse, abandonment, and other undesirable consequences like:-

- Excessive use of computer jargon and acronyms
- No obvious or less-than-intuitive design
- Inability to distinguish between alternative actions ("what do I do next?")
- Inconsistent problem-solving approaches
- Design inconsistency

1.9.1 Commandments of User Interface Design:

- Understand your users and their tasks.
- Involve the user in interface design.
- Test the system on actual users.
- Practice iterative design

1.9.2 Human Engineering Guidelines:

- The system user should always be aware of what to do next.
- Tell the user what the system expects right now.
- Tell the user that data has been entered correctly.
- Tell the user that data has not been entered correctly.
- Explain to the user the reason for a delay in processing.
- Tell the user that a task was completed or was not completed .
- The screen should be formatted so that the various types of information, instructions, and messages always appear in the same general display area.
- Messages, instructions, or information should be displayed long enough to allow the system user to read them.
- Use display attributes sparingly.
- Default values for fields and answers to be entered by the user should be specified
- Anticipate the errors users might make.
- With respect to errors, a user should not be allowed to proceed without correcting an

error.

- If the user does something that could be catastrophic, the keyboard should be locked to prevent any further input, and an instruction to call the analyst or technical support should be displayed.

1.9.3 Guidelines for dialogue Tone and Terminology Tone:

-Use simple, grammatically correct sentences.

-Don't be funny or cute.

-Don't be condescending .

Terminology:

-Don't use computer jargon.

-Avoid most abbreviations.

-Use simple terms.

-Be consistent in your use of terminology.

-Carefully phrase instructions use appropriate action verbs.

1.9.4 Common Approaches to Showing the Display Area:

- Paging: -displays a complete screen of characters at a time. The complete display area is known as a page (or screen). The page is replaced on demand by the next or previous page, much like turning the pages of a book.
- Scrolling: - moves the displayed information up or down on the screen, one line at a time. This is similar to the way movie and television credits scroll up the screen at the end of a movie.

1.9.5 Styles or Strategies Used For Designing Graphical User interfaces:

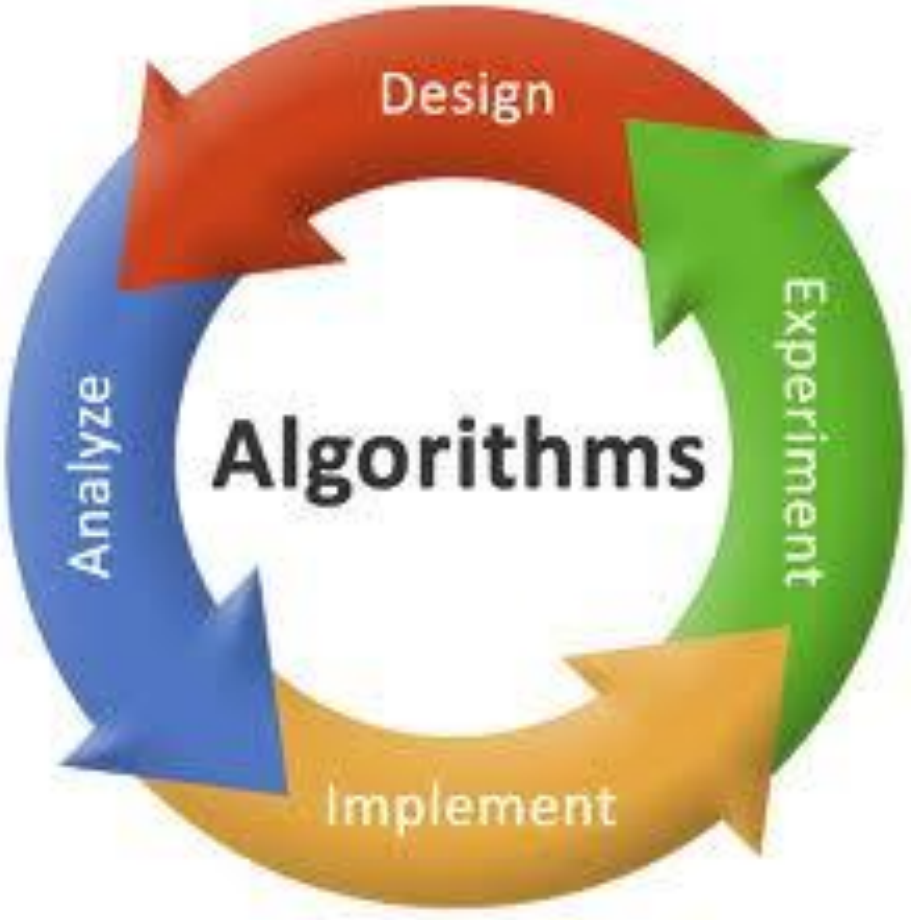
1-Windows and Frames.

2-Menu-Driven interfaces.

3-Instruction -Driven Interfaces.

4-Question-Answer Dialogue

chapter 2



2.1Before you begin:

Consider normalizing the pictures, if one is a higher resolution than the other, consider the option that one of them is a compressed version of the other, therefore scaling the resolution down might provide more accurate results.

Consider scanning various prospective areas of the image that could represent zoomed portions of the image and various positions and rotations. It starts getting tricky if one of the images are a skewed version of another, these are the sort of limitations you should identify and compromise on.

Matlab is an excellent tool for testing and evaluating images.

2.1.1 Testing the algorithms:

You should test (at the minimum) a large human analysed set of test data where matches are known beforehand. If for example in your test data you have 1,000 images where 5% of them match, you now have a reasonably reliable benchmark. An algorithm that finds 10% positives is not as good as one that finds 4% of positives in our test data. However, one algorithm may find all the matches, but also have a large 20% false positive rate, so there are several ways to rate your algorithms.

The test data should attempt to be designed to cover as many types of dynamics as possible that you would expect to find in the real world.

It is important to note that each algorithm to be useful must perform better than random guessing, otherwise it is useless to us!

You can then apply your software into the real world in a controlled way and start to analyse the results it produces. This is the sort of software project which can go on for infinitum, there are always tweaks and improvements you can make, it is important to bear that in mind when designing it as it is easy to fall into the trap of the never ending project.

2.1.2 Colour Buckets:

With two pictures, scan each pixel and count the colours. For example you might have the 'buckets':

```
white
red
blue
green
black
```

(Obviously you would have a higher resolution of counters). Every time you find a 'red' pixel, you increment the red counter. Each bucket can be representative of spectrum of colours, the higher resolution the more accurate but you should experiment with an acceptable difference rate.

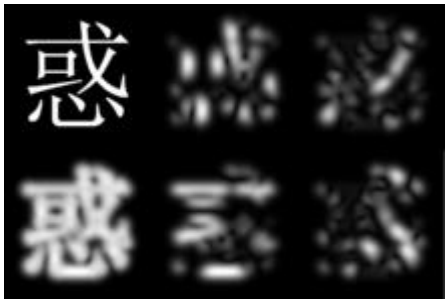
Once you have your totals, compare it to the totals for a second image. You might find that each image has a fairly unique footprint, enough to identify matches.

2.1.3 Edge detection:

How about using Edge Detection.

With two similar pictures edge detection should provide you with a usable and fairly reliable unique footprint.

Take both pictures, and apply edge detection. Maybe measure the average thickness of the edges and then calculate the probability the image could be scaled, and rescale if necessary. Below is an example of an applied Gabor Filter (a type of edge detection) in various rotations.



Compare the pictures pixel for pixel, count the matches and the non matches. If they are within a certain threshold of error, you have a match. Otherwise, you could try reducing the resolution up to a certain point and see if the probability of a match improves.

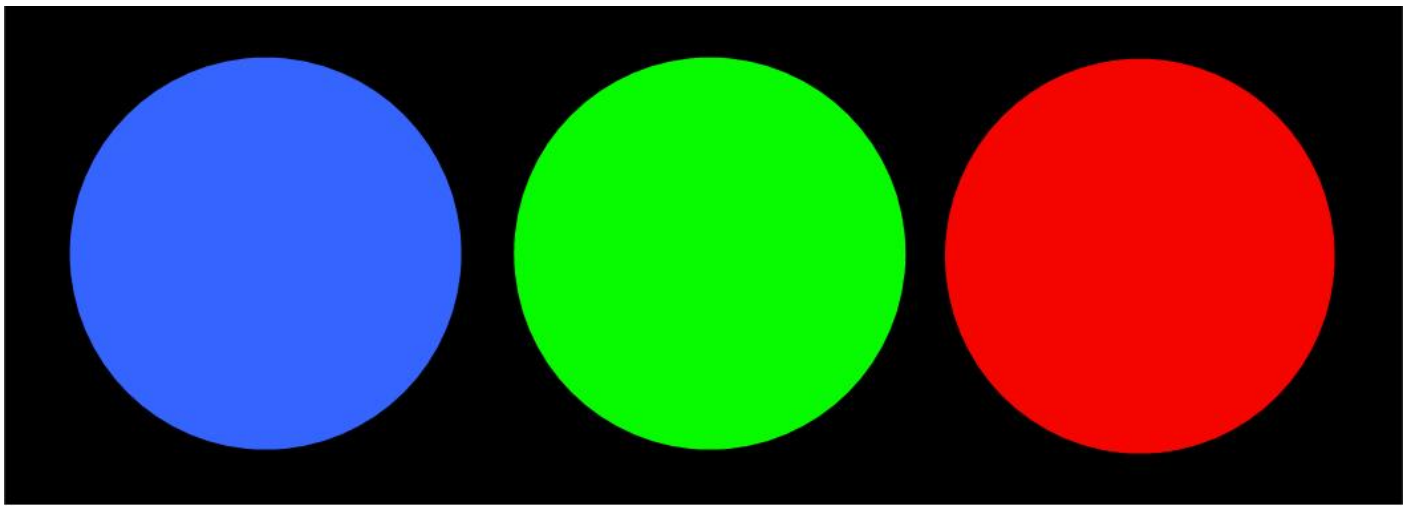
2.1.4 Regions of Interest:

Some images may have distinctive segments/regions of interest. These regions probably contrast highly with the rest of the image, and are a good item to search for in your other images to find matches.

Take this image for example:

The construction worker in blue is a region of interest and can be used as a search object. There are probably several ways you could extract properties/data from this region of interest and use them to search your data set.

If you have more than 2 regions of interest, you can measure the distances between them. Take this simplified example:



We have 3 clear regions of interest. The distance between region 1 and 2 may be 200 pixels, between 1 and 3 400 pixels, and 2 and 3 200 pixels.

Search other images for similar regions of interest, normalise the distance values and see if you have potential matches. This technique could work well for rotated and scaled images. The more regions of interest you have, the probability of a match increases as each distance measurement matches.

It is important to think about the context of your data set. If for example your data set is modern art, then regions of interest would work quite well, as regions of interest were probably *designed* to be a fundamental part of the final image. If however you are dealing with images of construction sites, regions of interest may be interpreted by the illegal copier as ugly and may be cropped/edited out liberally. Keep in mind common features of your

dataset, and attempt to exploit that knowledge.

2.1.5 Morphing :

Morphing two images is the process of turning one image into the other through a set of steps:

Note, this is different to fading one image into another!

There are many software packages that can morph images. It's traditionaly used as a transitional effect, two images don't morph into something halfway usually, one extreme morphs into the other extreme as the final result.

Why could this be useful? Dependant on the morphing algorithm you use, there may be a relationship between similarity of images, and some parameters of the morphing algorithm.

In a grossly over simplified example, one algorithm might execute faster when there are less changes to be made. We then know there is a higher probability that these two images share properties with each other.

This technique *could* work well for rotated, distorted, skewed, zoomed, all types of copied images. Again this is just an idea I have had, it's not based on any researched academia as far as I am aware (I haven't look hard though), so it may be a lot of work for you with limited/no results.

2.1.6 Zipping :

Ow's answer in this question is excellent, I remember reading about these sort of techniques studying AI. It is quite effective at comparing corpus lexicons.

One interesting optimisation when comparing corpuses is that you can remove words considered to be too common, for example 'The', 'A', 'And' etc. These words dilute our result, we want to work out how different the two corpus are so these can be removed before processing. Perhaps there are similar common signals in images that could be stripped before compression? It might be worth looking into.

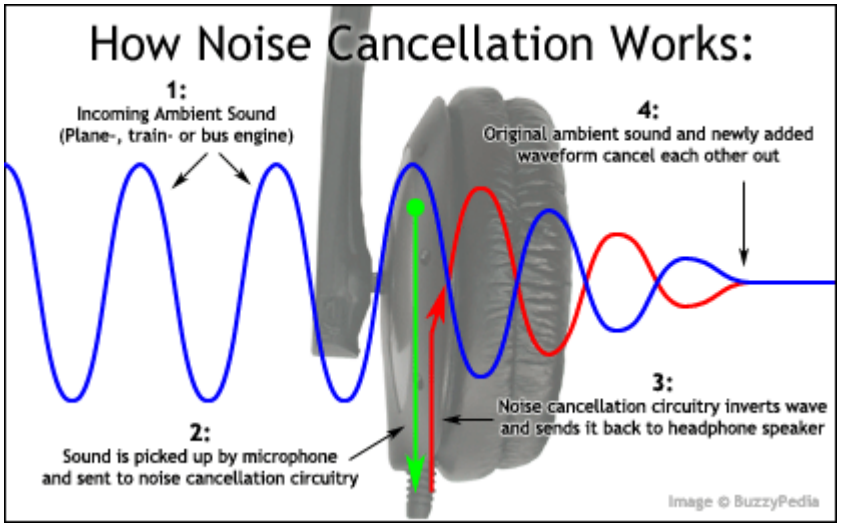
Compression ratio is a very quick and reasonably effective way of determining how similar two sets of data are. Reading up about how compression works will give you a good idea why this could be so effective. For a fast to release algorithm this would probably be a good starting point.

2.1.7 Transparency :

Again I am unsure how transparency data is stored for certain image types, gif png etc, but this will be extractable and would serve as an effective simplified cut out to compare with your data sets transparency.

2.1.8 Inverting Signals:

An image is just a signal. If you play a noise from a speaker, and you play the opposite noise in another speaker in perfect sync at the exact same volume, they cancel each other out.



Invert on of the images, and add it onto your other image. Scale it/loop positions repetitively until you find a resulting image where enough of the pixels are white (or black? I'll refer to it as a neutral canvas) to provide you with a positive match, or partial match.

However, consider two images that are equal, except one of them has a brighten effect applied to it:

Inverting one of them, then adding it to the other will not result in a neutral canvas which is what we are aiming for. However, when comparing the pixels from both original images, we can definatly see a clear relationship between the two.

I haven't studied colour for some years now, and am unsure if the colour spectrum is on a

linear scale, but if you determined the average factor of colour difference between both pictures, you can use this value to normalise the data before processing with this technique.

2.1.9Tree Data structures:

At first these don't seem to fit for the problem, but I think they could work.

You could think about extracting certain properties of an image (for example colour bins) and generate ahuffman tree or similar data structure. You might be able to compare two trees for similarity. This wouldn't work well for photographic data for example with a large spectrum of colour, but cartoons or other reduced colour set images this might work.

This probably wouldn't work, but it's an idea. The trie datastructure is great at storing lexicons, for example a dictionarty. It's a prefix tree. Perhaps it's possible to build an image equivalent of a lexicon, (again I can only think of colours) to construct a trie. If you reduced say a 300x300 image into 5x5 squares, then decompose each 5x5 square into a sequence of colours you could construct a trie from the resulting data. If a 2x2 square contains:

```
FFFFFF|000000|FDFD44|FFFFFF
```

We have a fairly unique trie code that extends 24 levels, increasing/decreasing the levels (IE reducing/increasing the size of our sub square) may yield more accurate results.

Comparing trie trees should be reasonably easy, and could possible provide effective results.

2.1.10 More ideas:

I stumbled accross an interesting paper breif about classification of satellite imagery, it outlines:
Texture measures considered are: cooccurrence matrices, gray-level differences, texture-tone analysis, features derived from the Fourier spectrum, and Gabor filters. Some Fourier features and some Gabor filters were found to be good choices, in particular when a single frequency band was used for classification.

It may be worth investigating those measurements in more detail, although some of them may not be relevant to your data set.

2.1.11 Other things to consider:

There are probably a lot of papers on this sort of thing, so reading some of them should help although they can be very technical. It is an extremely difficult area in computing, with many fruitless hours of work spent by many people attempting to do similar things. Keeping it simple and building upon those ideas would be the best way to go. It should be a reasonably difficult challenge to create an algorithm with a better than random match rate, and to start improving on that really does start to get quite hard to achieve.

Each method would probably need to be tested and tweaked thoroughly, if you have any information about the type of picture you will be checking as well, this would be useful. For example advertisements, many of them would have text in them, so doing text recognition would be an easy and probably very reliable way of finding matches especially when combined with other solutions. As mentioned earlier, attempt to exploit common properties of your data set.

Combining alternative measurements and techniques each that can have a weighted vote (dependant on their effectiveness) would be one way you could create a system that generates more accurate results.

If employing multiple algorithms, as mentioned at the begining of this answer, one may find all the positives but have a false positive rate of 20%, it would be of interest to study the properties/strengths/weaknesses of other algorithms as another algorithm may be effective in eliminating false positives returned from another.

2.2Image comparison algorithms:

- **Problem:**

  I'm looking to create a base table of images and then compare any new images against that to determine if the new image is an exact (or close) duplicate of the of the base. For example: if you want to reduce storage of the same image 100's of times, you could store one copy of it and provide reference links to it. When a new image is entered you want to compare to an existing image to make sure it's not a dup...ideas?

  (one of mine was to reduce to a small thumbnail and then randomly pick 100 pixel locations and compare...

- **Answer:**

  Below are three approaches to solving this problem (and there are many others).

- The first is a standard approach in computer vision, keypoint matching. This may require some background knowledge to implement, and can be slow.
- The second method uses only elementary image processing, and is potentially faster than the first approach, and is straightforward to implement. However, what it gains in understandability, it lacks in robustness -- matching fails on scaled, rotated, or discolored images.
- The third method is both fast and robust, but is potentially the hardest to implement.

2.2.1 Keypoint Matching:

Better than picking 100 random points is picking 100 *important* points. Certain parts of an image have more information than others (particularly at edges and corners), and these are the ones you'll want to use for smart image matching. Google "keypoint extraction" and "keypoint matching" and you'll find quite a few academic papers on the subject. These days, SIFT keypoints are arguably the most popular, since they can match images under different scales, rotations, and lighting. Some SIFT implementations can be found here. One downside to keypoint matching is the running time of a naive implementation: O(n^2m), where n is the number of keypoints in each image, and m is the number of images in the database. Some clever algorithms might find the closest match faster, like quadtrees or binary space partitioning.

2.2.2 Alternative solution: Histogram method:

Another less robust but potentially faster solution is to build feature histograms for each image, and choose the image with the histogram closest to the input image's histogram. I implemented this as an undergrad, and we used 3 color histograms (red, green, and blue), and two texture histograms, direction and scale. I'll give the details below, but I should note that this only worked well for matching images VERY similar to the database images. Re-scaled, rotated, or discolored images can fail with this method, but small changes like cropping won't break the algorithm

Computing the color histograms is straightforward -- just pick the range for your histogram buckets, and for each range, tally the number of pixels with a color in that range. For example, consider the "green" histogram, and suppose we choose 4 buckets for our histogram: 0-63, 64-127, 128-191, and 192-255. Then for each pixel, we look at the green value, and add a tally to the appropriate bucket. When we're done tallying, we divide each bucket total by the number of pixels in the entire image to get a normalized histogram for the green channel.

For the texture direction histogram, we started by performing edge detection on the image. Each edge point has a normal vector pointing in the direction perpendicular to the edge. We quantized the normal vector's angle into one of 6 buckets between 0 and PI (since edges have 180-degree symmetry, we converted angles between -PI and 0 to be between 0 and PI). After tallying up the number of edge points in each direction, we have an un-normalized histogram representing texture direction, which we normalized by dividing each bucket by the total number of edge points in the image.

To compute the texture scale histogram, for each edge point, we measured the distance to the next-closest edge point with the same direction. For example, if edge point A has a direction of 45 degrees, the algorithm walks in that direction until it finds another edge point with a direction of 45 degrees (or within a reasonable deviation). After computing this distance for each edge point, we dump those values into a histogram and normalize it by dividing by the total number of edge points.

Now you have 5 histograms for each image. To compare two images, you take the absolute value of the difference between each histogram bucket, and then sum these values. For example, to compare images A and B, we would compute

```
|A.green_histogram.bucket_1 - B.green_histogram.bucket_1|
```

for each bucket in the green histogram, and repeat for the other histograms, and then sum up all the results. The smaller the result, the better the match. Repeat for all images in the database, and the match with the smallest result wins. You'd probably want to have a threshold, above which the algorithm concludes that no match was found.

2.2.3Third Choice - Keypoints + Decision Trees

A third approach that is probably much faster than the other two is using semantic texton forests (PDF). This involves extracting simple keypoints and using a collection decision trees to classify the image. This is faster than simple SIFT keypoint matching, because it avoids the costly matching process, and keypoints are much simpler than SIFT, so keypoint

extraction is much faster. However, it preserves the SIFT method's invariance to rotation, scale, and lighting, an important feature that the histogram method lacked.

2.2.4 The choosed algorithm Pixel grabber:

- PixelGrabber is an ImageConsumer that can collect pixel data into an array such that pixels can be processed in its entirety before sending it out again.
- Thus PixelGrabber marks the end of the push model and the beginning of an immediate mode model because the image data is put into an array instead of being passed to another ImageConsumer.
- Once you have pixel data in the originalPixelArray as in the above code, we can process it resulting in, say, newPixelArray.

2.2.5We can use it as:

```
PixelGrabber grabber = new PixelGrabber(originalImg,
                                                    0, 0, -1, -1,true);
try {
        if (grabber.grabPixels()) {
                int width = grabber.getWidth();
                int height = grabber.getHeight();
                int[] originalPixelArray = (int[]) grabber.getPixels();
        } else {
                System.err.println("Grabbing Failed");
        }
} catch (InterruptedException e) {
        System.err.println("PixelGrabbing interrupted");
}
```

2.2.6 The main class is in that format:

```
public void handlesinglepixel(int x, int y, int pixel) {
        int alpha = (pixel >> 24) & 0xff;
        int red   = (pixel >> 16) & 0xff;
        int green = (pixel >>  8) & 0xff;
        int blue  = (pixel      ) & 0xff;
        // Deal with the pixel as necessary...
 }

 public void handlepixels(Image img, int x, int y, int w, int h) {
        int[] pixels = new int[w * h];
        PixelGrabber pg = new PixelGrabber(img, x, y, w, h, pixels, 0, w);
        try {
            pg.grabPixels();
```

```
        } catch (InterruptedException e) {
            System.err.println("interrupted waiting for pixels!");
            return;
        }
        if ((pg.getStatus() & ImageObserver.ABORT) != 0) {
            System.err.println("image fetch aborted or errored");
            return;
        }
        for (int j = 0; j < h; j++) {
            for (int i = 0; i < w; i++) {
                handlesinglepixel(x+i, y+j, pixels[j * w + i]);
            }
        }
    }
```

2.3 Image Filtering algorithms:

2.3.1 Introduction

Image filtering allows you to apply various effects on photos. The type of image filtering described here uses a 2D filter similar to the one included in Paint Shop Pro as User Defined Filter and in Photoshop as Custom Filter.

2.3.2Convolution:

The trick of image filtering is that you have a 2D filter matrix, and the 2D image. Then, for every pixel of the image, take the sum of products. Each product is the color value of the current pixel or a neighbor of it, with the corresponding value of the filter matrix. The center of the filter matrix has to be multiplied with the current pixel, the other elements of the filter matrix with corresponding neighbor pixels.

This operation where you take the sum of products of elements from two 2D functions, where you let one of the two functions move over every element of the other function, is called Convolution or Correlation. The difference between Convolution and Correlation is that for Convolution you have to mirror the filter matrix, but usually it's symmetrical anyway so there's no difference.

The filters with convolution are relatively simple. More complex filters, that can use more fancy functions, exist as well, and can do much more complex things (for example the Colored Pencil filter in Photoshop), but such filters aren't discussed here.

The 2D convolution operation requires a 4-double loop, so it isn't extremely fast, unless you use small filters. Here we'll usually be using 3x3 or 5x5 filters.

There are a few rules about the filter:

- It's size has to be uneven, so that it has a center, for example 3x3, 5x5 and 7x7 are ok.
- It doesn't have to, but the sum of all elements of the filter should be 1 if you want the resulting image to have the same brightness as the original.
- If the sum of the elements is larger than 1, the result will be a brighter image, and if it's smaller than 1, a darker image. If the sum is 0, the resulting image isn't necessarily completely black, but it'll be very dark.

The image has finite dimensions, and if you're for example calculating a pixel on the left side, there are no more pixels to the left of it while these are required for the convolution. You can either use value 0 here, or wrap around to the other side of the image. In this tutorial, the wrapping around is chosen because it can easily be done with a modulo division.

The resulting pixel values after applying the filter can be negative or larger than 255, if that happens you can truncate them so that values smaller than 0 are made 0 and values larger than 255 are set to 255. For negative values, you can also take the absolute value instead.

In the Fourier Domain or Frequency Domain, the convolution operation becomes a multiplication instead, which is faster. In the Fourier Domain, much more powerful and bigger filters can be applied faster, especially if you use the Fast Fourier Transform. More about this is in the Fourier Transform article. In this article, we'll look at a few very typical small filters, such as blur, edge detection and emboss.

Image filters aren't feasible for real time applications and games yet, but they're useful in image processing.

Digital audio and electronic filters work with convolution as well, but in 1D.

Here's the code that'll be used to try out different filters. Apart from using a filter matrix, it also has a multiplier factor and a bias. After applying the filter, the factor will be multiplied with the result, and the bias added to it. So if you have a filter with an element 0.25 in it, but the factor is set to 2, all elements of the filter are in theory multiplied by two so that element 0.25 is actually 0.5. The bias can be used if you want to make the resulting image brighter.

The result of one pixel is stored in floats red, green and blue, before converting it to the integer value in the result buffer.

The filter calculation itself is a 4-double loop that has to go through every pixel of the image, and then through every element of the filter matrix. The location imageX and imageY is

calculated so that for the center element of the filter it'll be x, y, but for the other elements it'll be a pixel from the image to the left, right, top or bottom of x, y. It's modulo divided through the width (w) or height (h) of the image so that pixels outside the image will be wrapped around. Before modulo dividing it, w or h are also added to it, because this modulo division doesn't work correctly for negative values. Now, pixel (-1, -1) will correctly become pixel (w-1, h-1).

```
#define filterWidth 3
#define filterHeight 3
#define imageWidth 320
#define imageHeight 240
//declare image buffers
ColorRGB image[imageWidth][imageHeight];
ColorRGB result[imageWidth][imageHeight];

double filter[filterWidth][filterHeight] =
{
     0, 0, 0,
     0, 1, 0,
     0, 0, 0
};

double factor = 1.0;
double bias = 0.0;

int main(int argc, char *argv[])
{
    //set up the screen
    screen(imageWidth, imageHeight, 0, "Filters");

    //load the image into the buffer
    loadBMP("pics/photo3.bmp", image[0], imageWidth , imageHeight);

    //apply the filter
    for(int x = 0; x < w; x++)
    for(int y = 0; y < h; y++)
    {
        double red = 0.0, green = 0.0, blue = 0.0;

        //multiply every value of the filter with corresponding image pixel
        for(int filterX = 0; filterX < filterWidth; filterX++)
        for(int filterY = 0; filterY < filterHeight; filterY++)
        {
            int imageX = (x - filterWidth / 2 + filterX + w) % w;
            int imageY = (y - filterHeight / 2 + filterY + h) % h;
            red += image[imageX][imageY].r * filter[filterX][filterY];
            green += image[imageX][imageY].g * filter[filterX][filterY];
            blue += image[imageX][imageY].b * filter[filterX][filterY];
        }

        //truncate values smaller than zero and larger than 255
        result[x][y].r = min(max(int(factor * red + bias), 0), 255);
        result[x][y].g = min(max(int(factor * green + bias), 0), 255);
        result[x][y].b = min(max(int(factor * blue + bias), 0), 255);
    }

    //draw the result buffer to the screen
    for(int x = 0; x < w; x++)
    for(int y = 0; y < h; y++)
    {
        pset(x, y, result[x][y]);
    }

    //redraw & sleep
    redraw();
    sleep();
}
```

If you want to take the absolute value of values smaller than zero instead of truncating it, use this code instead:

```
//take absolute value and truncate to 255

        result[x][y].r = min(abs(int(factor * red + bias)), 255);
        result[x][y].g = min(abs(int(factor * green + bias)), 255);
        result[x][y].b = min(abs(int(factor * blue + bias)), 255);
```

The filter filled in currently,

[0 0 0]
[0 1 0]
[0 0 0],

does nothing more than returning the original image, since only the center value is 1 so every pixel is multiplied with 1.
The original image looks like this:

Now we'll apply several filters to the image by changing the definition of the filter array and running the code.

2.3.3 Blur:

Blurring is done for example by taking the average of the current pixel and it's 4 neighbors. Take the sum of the current pixel and it's 4 neighbors, and divide it through 5, or thus fill in 5 times the value 0.2 in the filter:

```
double filter[filterWidth][filterHeight] =
{
     0.0, 0.2,  0.0,
     0.2, 0.2,  0.2,
     0.0, 0.2,  0.0
};

double factor = 1.0;
double bias = 0.0;
```

With such a small filter matrix, this gives only a very soft blur:

With a bigger filter you can blur it a bit more (don't forget to change the filterWidth and filterHeight values):

```
double filter[filterWidth][filterHeight] =
{
    0, 0, 1, 0, 0,
    0, 1, 1, 1, 0,
    1, 1, 1, 1, 1,
    0, 1, 1, 1, 0,
    0, 0, 1, 0, 0,
};

double factor = 1.0 / 13.0;
double bias = 0.0;
```

The sum of all elements of the filter should be 1, but instead of filling in some floating point value inside the filter, instead the factor is divided through the sum of all elements, which is 13.

This blurs it a bit more already:

The more blur you want, the bigger the filter has to be, or you can apply the same small blur filter multiple times.

2.3.4 Motion Blur

Motion blur is achieved by blurring in only 1 direction. Here's a 9x9 motion blur filter:

```
double filter[filterWidth][filterHeight] =
{
    1, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 1, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 1,
};

double factor = 1.0 / 9.0;
double bias = 0.0;
```

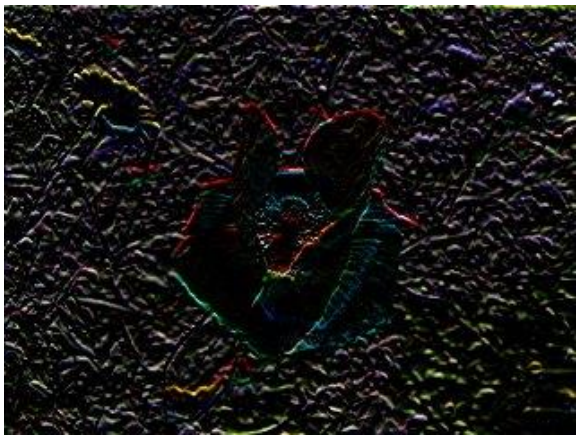
It's as if the camera is moving from the top left to the bottom right, hence the name.

2.3.5 Find Edges:

A filter to find the horizontal edges can look like this:

```
double filter[filterWidth][filterHeight] =
{
     0,  0,  0,  0,  0,
     0,  0,  0,  0,  0,
    -1, -1,  2,  0,  0,
     0,  0,  0,  0,  0,
     0,  0,  0,  0,  0,
};

double factor = 1.0;
double bias = 0.0;
```

A filter of 5x5 instead of 3x3 was chosen, because the result of a 3x3 filter is too dark on the current image. Note that the sum of all the elements is 0 now, which will result in a very dark image where only the edges it detected are colored.



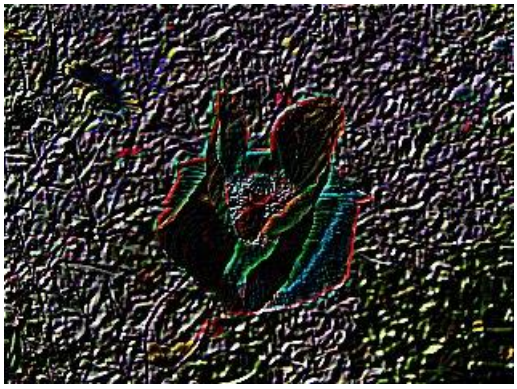
The reason why this filter can find horizontal edges, is that the convolution operation with this filter can be seen as a sort of discrete version of the derivative: you take the current pixel and subtract the value of the previous one from it, so you get a value that represents the difference between those two or the slope of the function.

Here's a filter that'll find vertical edges instead, and uses both pixel values below and above the current pixel:

```
double filter[filterWidth][filterHeight] =
{
     0,  0, -1,  0,  0,
     0,  0, -1,  0,  0,
     0,  0,  4,  0,  0,
     0,  0, -1,  0,  0,
     0,  0, -1,  0,  0,
};

double factor = 1.0;
double bias = 0.0;
```

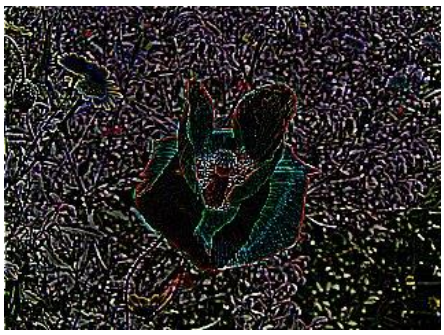
Here's yet another possible filter, one that's good at finding edges of 45°. The values '-2' were chosen for no particular reason at all, just make sure the sum of the values is 0.

```
double filter[filterWidth][filterHeight] =
{
    -1,  0,  0,  0,  0,
     0, -2,  0,  0,  0,
     0,  0,  6,  0,  0,
     0,  0,  0, -2,  0,
     0,  0,  0,  0, -1,
};

double factor = 1.0;
double bias = 0.0;
```



And here's a simple edge detection filter that detects edges in all directions:

```
double filter[filterWidth][filterHeight] =
{
    -1, -1, -1,
    -1,  8, -1,
    -1, -1, -1
};

double factor = 1.0;
double bias = 0.0;
```



2.3.6 Sharpen:

To sharpen the image is very similar to finding edges, add the original image, and the image after the edge detection to each other, and the result will be a new image where the edges are enhanced, making it look sharper. Adding those two images is done by taking the edge detection filter from the previous example, and incrementing the center value of it with 1. Now the sum of the filter elements is 1 and the result will be an image with the same

brightness as the original, but sharper.

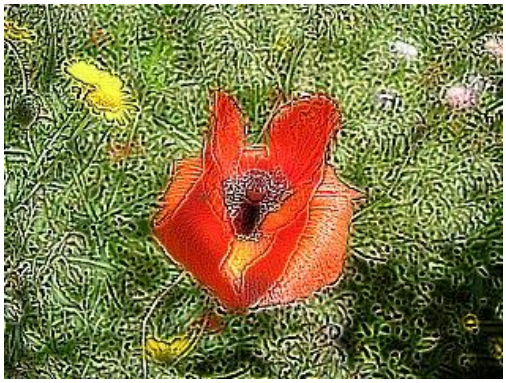
```
double filter[filterWidth][filterHeight] =
{
    -1, -1, -1,
    -1,  9, -1,
    -1, -1, -1
};

double factor = 1.0;
double bias = 0.0;
```

Here's a more subtle sharpen filter:

```
double filter[filterWidth][filterHeight] =
{
    -1, -1, -1, -1, -1,
    -1,  2,  2,  2, -1,
    -1,  2,  8,  2, -1,
    -1,  2,  2,  2, -1,
    -1, -1, -1, -1, -1,
};

double factor = 1.0 / 8.0;
double bias = 0.0;
```

Here's a filter that shows the edges excessively:

```
double filter[filterWidth][filterHeight] =
{
     1,  1,  1,
     1, -7,  1,
     1,  1,  1
};

double factor = 1.0;
double bias = 0.0;
```

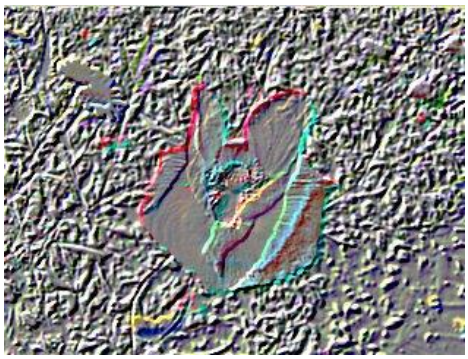


2.3.7 Emboss:

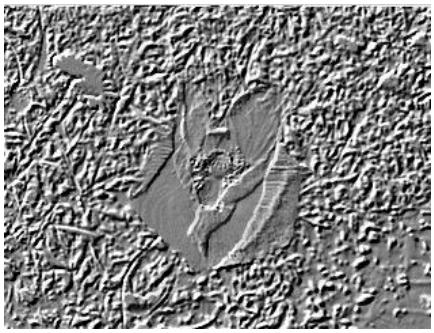
An emboss filter gives a 3D shadow effect to the image, the result is very useful for a bumpmap of the image. It can be achieved by taking a pixel on one side of the center, and subtracting one of the other side from it. Pixels can get either a positive or a negative result. To use the negative pixels as shadow and positive ones as light, for a bumpmap, a bias of 128 is added to the image. Now, most parts of the image will be gray, and the sides will be either dark gray/black or bright gray/white.

For example here's an emboss filter with an angle of 45°:

```
double filter[filterWidth][filterHeight] =
{
    -1, -1,  0,
    -1,  0,  1,
     0,  1,  1
};

double factor = 1.0;
double bias = 128.0;
```

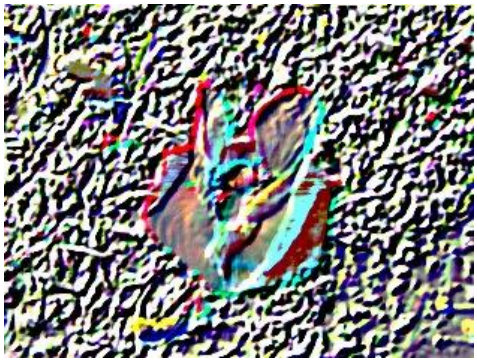


If you really want to use it as bumpmap, grayscale it:



Here's a much more exaggerated emboss filter:

```
double filter[filterWidth][filterHeight] =
{
    -1, -1, -1, -1,  0,
    -1, -1, -1,  0,  1,
    -1, -1,  0,  1,  1,
    -1,  0,  1,  1,  1,
     0,  1,  1,  1,  1
};

double factor = 1.0;
double bias = 128.0;
```



2.3.8 Mean and Median Filter:

Both the Mean Filter and the Median Filter can be used to remove noise from an image. A Mean Filter is a filter that takes the average of the current pixel and it's neighbors, for example if you use it's 8 neighbors it becomes the filter with kernel:

```
double filter[filterWidth][filterHeight] =
{
    1, 1, 1,
    1, 1, 1,
    1, 1, 1
};

double factor = 1.0 / 9.0;
double bias = 0.0;
```

This is an ordinary blur filter. We can test it on the following image with so called "Salt and Pepper" Noise:

When applied, it gives a blurry result:

The Median Filter does somewhat the same, but, instead of taking the mean or average, it takes the median. The median is gotten by sorting all the values from low to high, and then taking the value in the center. If there are two values in the center, the average of these two is taken. A median filter gives better results to remove salt and pepper noise, because it completely eliminates the the noise. With an average filter, the color value of the noise particles are still used in the average calculations, when taking the median you only keep the color value of one or two healthy pixels. The median filter also reduces the image quality however.

Such a median filter can't be done with a convolution, and a sorting algorithm is needed, in this case combsort was chosen, which is a relatively fast sorting algorithm.

To get the median of the current pixel and it's 8 neighbors, set filterWidth and filterHeight to

3, but you can also make it higher to remove larger noise particles.

Here are the variable declarations. The arrays red, green and blue will contain the values of the current pixel and all of it's neighbors, and these are the arrays that'll be sorted by the sorting algorithm to be able to take the median value.

```
#define filterWidth 3
#define filterHeight 3
#define imageWidth 320
#define imageHeight 240

//image buffers
ColorRGB image[imageWidth][imageHeight];
ColorRGB result[imageWidth][imageHeight];

//color arrays
int red[filterWidth * filterHeight];
int green[filterWidth * filterHeight];
int blue[filterWidth * filterHeight];

void combsort(int * data, int amount);
```

The main function applies the filter, calculates the medians and then draws the result.

```
int main(int argc, char *argv[])
{ een
    screen(imageWidth, imageHeight, 0, "Median Filter");   //load the image into the buffer
    loadBMP("pics/noise.bmp", image[0], imageWidth , imageHeight);

    //apply the filter
    for(int x = 0; x < w; x++)
    for(int y = 0; y < h; y++)
    {
        int n = 0;
        //set the color values in the arrays
        for(int filterX = 0; filterX < filterWidth; filterX++)
        for(int filterY = 0; filterY < filterHeight; filterY++)
        {
            int imageX = (x - filterWidth / 2 + filterX + w) % w;
            int imageY = (y - filterHeight / 2 + filterY + h) % h;
            red[n] = image[imageX][imageY].r;
            green[n] = image[imageX][imageY].g;
            blue[n] = image[imageX][imageY].b;
            n++;
        }

        combsort(red, filterWidth * filterHeight);
        combsort(green, filterWidth * filterHeight);
        combsort(blue, filterWidth * filterHeight);

        //take the median, if the length of the array is even, take the average of both
center values
        if((filterWidth * filterHeight) % 2 == 1)
        {
            result[x][y].r = red[filterWidth * filterHeight / 2];
            result[x][y].g = green[filterWidth * filterHeight / 2];
            result[x][y].b = blue[filterWidth * filterHeight / 2];
        }
        else if(filterWidth >= 2)
        {
            result[x][y].r = (red[filterWidth * filterHeight / 2] + red[filterWidth *
filterHeight / 2 + 1]) / 2;
            result[x][y].g = (green[filterWidth * filterHeight / 2] + green[filterWidth *
filterHeight / 2 + 1]) / 2;
            result[x][y].b = (blue[filterWidth * filterHeight / 2] + blue[filterWidth *
filterHeight / 2 + 1]) / 2;
        }
    }

    //draw the result buffer to the screen
    for(int x = 0; x < w; x++)
    for(int y = 0; y < h; y++)
    {
        pset(x, y, result[x][y]);
    }

    //redraw & sleep
    redraw();
    sleep();
}
```

The combsort function sorts the arrays of color values. The array contains the value of every color in the rectangular area you're working on, but it's not sorted so you can't immediatly take the median of it. This sorting algorithm orders it from low to high, so that you know that the center value of the sorted array will be the median. Combsort is a modified version of bubblesort, that uses a gap starting at the size of the array and shrinking by 1.3 each step, this was found to be the optimal shrink value. Setting gap to 11 if it's 9 or 10 also makes it faster. This quickly eliminates "turtles", which are low values at the sides, known to make bubble sort slow. Finally, when gap is 1, it acts like a normal bubble sort, but will go very fast because most turtles are gone. It's not as fast as quicksort, but close, and much simpler to

write.

```
//combsort: bubble sort made faster by using gaps to eliminate turtles
void combsort(int * data, int amount)
{
    int gap = amount;
    bool swapped = false;
    while(gap > 1 || swapped)
    {
        //shrink factor 1.3
        gap = (gap * 10) / 13;
        if(gap == 9 || gap == 10) gap = 11;
        if (gap < 1) gap = 1;
        swapped = false;
        for (int i = 0; i < amount - gap; i++)
        {
            int j = i + gap;
            if (data[i] > data[j])
            {
                data[i] += data[j];
                data[j] = data[i] - data[j];
                data[i] -= data[j];
                swapped = true;
            }
        }
    }
}
```

Here's again the noisy image:

The 3x3 median filter removes it's noise:

Higher sizes of filters go pretty slow, because the code is very unoptimized. Their results are somewhat artistic though. Here is the result of different sizes:

5x5:

9x9:

15x15:

2.3.8 Conclusion:

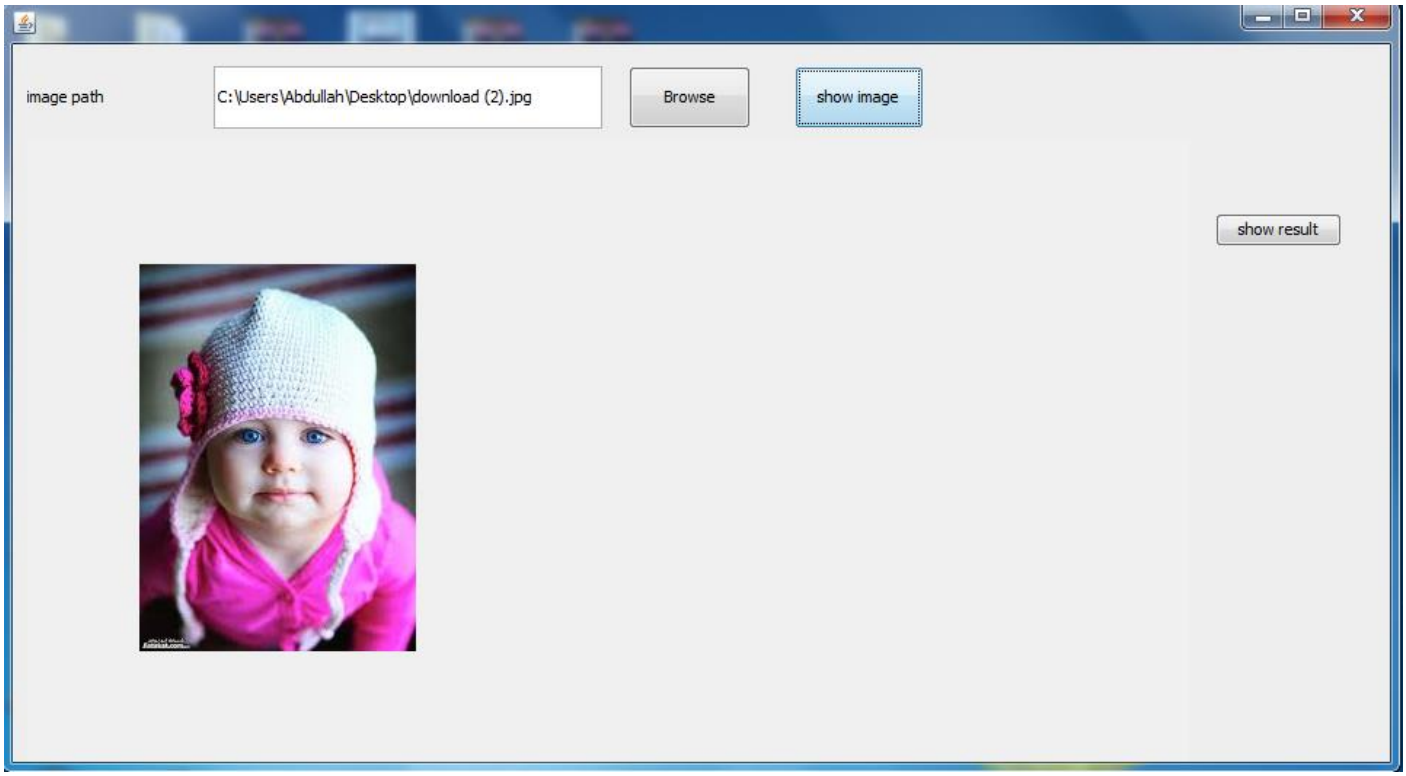
This article contained code to apply convolution filters on images, and showed a few different filters and their result. These are only the very basics of image filtering, with bigger filters and a lot of tweaking you can get much better filters.

The Fourier Transform article shows a different way to filter images, in the frequency domain. There Low Pass, High Pass and Band Pass filters are discussed.

CHAPTER 3

IMPLEMENTATION

3.1interface implementation:




```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;


public class browse extends javax.swing.JFrame {
BufferedImage image;
 int array[][];

    public browse() {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(browse.class.getName()).log(Level.SEVERE, null, ex);
        } catch (InstantiationException ex) {
            Logger.getLogger(browse.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IllegalAccessException ex) {
            Logger.getLogger(browse.class.getName()).log(Level.SEVERE, null, ex);
        } catch (UnsupportedLookAndFeelException ex) {
            Logger.getLogger(browse.class.getName()).log(Level.SEVERE, null, ex);
        }
```

```
}

    private void jButton2StateChanged(javax.swing.event.ChangeEvent evt) {
        // TODO add your handling code here:
    }

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                new browse().setVisible(true);
            }
        });
    }
    // Variables declaration - do not modify
    private javax.swing.JButton jButton1;
    private javax.swing.JButton jButton2;
    private javax.swing.JButton jButton3;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JTextField jTextField1;
    private javax.swing.JLabel lbl;
    private javax.swing.JPanel p;
    // End of variables declaration
}
```

3.2 comparison implementation:

we choosed a simple comparison algorithm (Pixel Grabber) Algorithm and we had been talken about it before now we will talk about it by its coding usage in full code:

```
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.PixelGrabber;

public class Compare {

static void processImage() {

String file1 = "img1.png";
String file2 = "img2.png";

Image image1 = Toolkit.getDefaultToolkit().getImage(file1);
Image image2 = Toolkit.getDefaultToolkit().getImage(file2);

try {

PixelGrabber grab1 =new PixelGrabber(image1, 0, 0, -1, -1, false);
PixelGrabber grab2 =new PixelGrabber(image2, 0, 0, -1, -1, false);

int[] data1 = null;

if (grab1.grabPixels()) {
int width = grab1.getWidth();
int height = grab1.getHeight();
data1 = new int[width * height];
data1 = (int[]) grab1.getPixels();
}

int[] data2 = null;

if (grab2.grabPixels()) {
int width = grab2.getWidth();
int height = grab2.getHeight();
data2 = new int[width * height];
data2 = (int[]) grab2.getPixels();
}

System.out.println("Pixels equal: " + java.util.Arrays.equals(data1, data2));

} catch (InterruptedException e1) {
e1.printStackTrace();
}
}
```

```
public static void main(String args[]) {
processImage();
}
}
```

3.4 filtering implementation:

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BandCombineOp;
import java.awt.image.BufferedImage;
import java.awt.image.BufferedImageOp;
import java.awt.image.ByteLookupTable;
import java.awt.image.ConvolveOp;
import java.awt.image.Kernel;
import java.awt.image.LookupOp;
import java.awt.image.Raster;
import java.awt.image.WritableRaster;
import java.net.URL;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Java2DExample extends JFrame {
  private JMenu filterMenu = new JMenu("Image Filters");
  private ImagePanel imagePanel;
  private MyFilter invertFilter = new InvertFilter();
  private MyFilter sharpenFilter = new SharpenFilter();
  private MyFilter blurFilter = new BlurFilter();
  private MyFilter colorFilter = new ColorFilter();
  public Java2DExample() {
    super("Java 2D Image Processing Demo");
    imagePanel = new ImagePanel(Java2DExample.class.getResource("yourI
mage.png"));
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);
    filterMenu.setMnemonic('I');
    JMenuItem originalMenuItem = new JMenuItem("Display Original");
    originalMenuItem.setMnemonic('O');
    originalMenuItem.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent action) {
        imagePanel.displayOriginalImage();
}
    });
```

```
JMenuItem invertMenuItem = createMenuItem("Invert", 'I', invertFilter)
;
    JMenuItem sharpenMenuItem = createMenuItem("Sharpen", 'S', sharpen
Filter);
    JMenuItem blurMenuItem = createMenuItem("Blur", 'B', blurFilter);
    JMenuItem changeColorsMenuItem = createMenuItem("Change Colors", '
C', colorFilter);
    filterMenu.add(originalMenuItem);
    filterMenu.add(invertMenuItem);
    filterMenu.add(sharpenMenuItem);
    filterMenu.add(blurMenuItem);
    filterMenu.add(changeColorsMenuItem);
    menuBar.add(filterMenu);
    getContentPane().add(imagePanel, BorderLayout.CENTER);
  }
  public JMenuItem createMenuItem(String menuItemName, char mnemonic,
final MyFilter filter) {
    JMenuItem menuItem = new JMenuItem(menuItemName);
    menuItem.setMnemonic(mnemonic);
    menuItem.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent action) {
        imagePanel.applyFilter(filter);
      }
    });
    return menuItem;
  }
  public static void main(String args[]) {
    Java2DExample application = new Java2DExample();
    application.setDefaultCloseOperation(EXIT_ON_CLOSE);
    application.pack();
    application.setVisible(true);
  }
}
interface MyFilter {
  public abstract BufferedImage processImage(BufferedImage image);
}
class BlurFilter implements MyFilter {
  public BufferedImage processImage(BufferedImage image) {
    float[] blurMatrix = { 1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f, 1.0f
 / 9.0f, 1.0f / 9.0f,
        1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f };
    BufferedImageOp blurFilter = new ConvolveOp(new Kernel(3, 3, blurM
atrix),
        ConvolveOp.EDGE_NO_OP, null);
    return blurFilter.filter(image, null);
  }
}
class ImagePanel extends JPanel {
  private BufferedImage displayImage;
  private BufferedImage originalImage;
  private Image image;
  public ImagePanel(URL imageURL) {
    image = Toolkit.getDefaultToolkit().createImage(imageURL);
    MediaTracker mediaTracker = new MediaTracker(this);
    mediaTracker.addImage(image, 0);
```

```
class ColorFilter implements MyFilter {
  public BufferedImage processImage(BufferedImage image) {
    float[][] colorMatrix = { { 1f, 0f, 0f }, { 0.5f, 1.0f, 0.5f }, { 0.2f, 0.4f, 0.6f } };
    BandCombineOp changeColors = new BandCombineOp(colorMatrix, null);
    Raster sourceRaster = image.getRaster();
    WritableRaster displayRaster = sourceRaster.createCompatibleWritableRaster();
    changeColors.filter(sourceRaster, displayRaster);
    return new BufferedImage(image.getColorModel(), displayRaster, true, null);
  }
}
```

Chapter

Conclusion & Future Work

4.1Conclusion:

Finally, we discussed to explain that Face Finder (FF) project in an abbreviated points :

- ***Goal*** : is a program to check images (faces of persons) in needed situation to do that like airports or police centers etc.
- ***Advantage*** :
    - solve the name confusion problem (mr.X problem).
    - Fast algorithm of detection persons.
    - Catch illegal not good persons.
    - Show the wonted men quickly.
- ***Disadvantages (honesty)*** :
    - Program is little slow .
    - Lot of results may cause confusions.
- ***Main steps:***
    - Analysis is the main key.
    - Design is the brain.
    - Implementation is the body.
    - Testing is the confirmation way.
    - Easy algorithm usually good.
    - Knowledge is the shortcut to the point.
    - Arranging steps is so good.

- ***Main ideas:***
    - ***Image processing*****: Image processing involves changing the nature of an image in order to either**

      1. improve its pictorial information for human interpretation,

      2. render it more suitable for autonomous machine perception.

    - **An *Image retrieval* system is a computer system for browsing, searching and retrieving images from a large database of digital images**
    - ***Image search:***

      is a specialized data search used to find images. To search for images, a user may provide query terms such as keyword, image

file/link, or click on some image, and the system will return images "similar" to the query. The similarity used for search criteria could be meta tags, color distribution in images, region/shape attributes, etc.

- **System Analysis:**

  In this phase, the current system is studied in detail. A person responsible for the analysis of the system is known as analyst. In system analysis, the analyst conducts the following activities.

- **System design:**

  Is the process or art of defining thearchitecture, components, modules, interfaces, and data for a system to satisfy specified requirements.

- Pixel grabber is good algorithm for comparison.

4.2 future work:

- We can develop the program to be face detection or face recognition program
- We may develop it to recognize the finger print.
- We can develop it to search the face in faces samples.