Image Deduplication (IMDEDU)

Master Special Project Report

Chair of Applied Mathematics / Faculty of Civil Engineering / Bauhaus-University Weimar

Olubunmi Emmanuel Ogunleye (126829)
Selva Ganapathy Elangovan (124769)

Examiners
Prof. Dr. rer. nat. Björn Rüffer
Dr. rer. nat. habil. Michael Schönlein

Date of submission: March 7, 2025

Bauhaus-Universität Weimar

Declaration

We hereby affirm that the work presented in this project report is our own and has been carried out under standard supervision. We confirm that any use of AI tools has been solely for grammar editing and corrections. Furthermore, we affirm that this project report contains no material that has been previously accepted or is currently under examination for the award of any other degree or diploma at any university or tertiary institution. To the best of our knowledge, this report does not contain any material published or written by another person, except where proper attribution has been provided. We consent to the final version of this project report being made publicly accessible through the University's digital repository, subject to relevant copyright laws and any approved embargo.

Weimar, March 7, 2025

Olubunmi Emmanuel Ogunleye (126829)
Selva Ganapathy Elangovan (124769)

Contents

1 Introduction

1.1 Background of Study

In the digital era, it is common to encounter a significant number of digital images, often with multiple versions of the same photo scattered across various storage locations on computers and external drives. These images are typically generated through activities such as documenting daily life, downloading similar images from online sources, or creating edited versions with modifications like cropping, down scaling, or applying aesthetic effects.

A practical example of this scenario can be seen in managing old family photo collections. Over time, multiple versions of the same image may accumulate, each reflecting a specific modification or purpose. While all these versions may hold sentimental value, retaining every single one can lead to disorganization and inefficient use of storage space. [Thyagharajan and Kalaiarasi, 2021] noted that detecting near duplicates of an image helps reduce the amount of data to be stored in data storage and reduces search complexity when trying to retrieve information. This challenge highlights the need for effective methods to organize and archive digital image collections while maintaining the most representative or high-quality version of each photo.

1.2 Aim and Objectives

The aim of this study is to devise an algorithm that detects near duplicates and groups pictures that are essentially showing the same contents into groups. In order to achieve this aim, the objectives are:

1. To implement a way to gather a collection of images from a set of hard drives along with meta information.
2. Investigate ways to visually define a metric to measure similarity between images.
3. Implement a technique to cluster groups of images that are visually close to each other.
4. Implement a measure to determine the best version of the group of similar images by resolution.

2 Literature Review

This research investigates methods for detecting near-duplicate images within a collection of pictures stored on a hard drive or within a designated directory. While advanced techniques like neural networks offer promising solutions, this study focuses on leveraging hash functions due to limitations in the available dataset. The near-duplicate detection process will be implemented using Python libraries, incorporating established principles such as hash functions, clustering algorithms, and similarity measures. Streamlit will be employed to design a user-friendly graphical user interface (GUI) that enhances the user experience.

To effectively detect near-duplicates, this research will encompass several key steps: reading images from the hard drive, computing cryptographic hashes for efficient image metadata storage, and utilizing perceptual hashes to assess image similarity and perform clustering. This will enable the identification of groups of visually similar images.

The subsequent sections will delve into the mathematical underpinnings of the employed hash functions, similarity measures, and clustering algorithms. Furthermore, detailed explanations of the utilized Python functions and libraries will be provided.

2.1 Hash Functions

Given the constraints of a limited training dataset, this study leverages a hashing algorithm to detect image duplicates. This chapter provides an in-depth discussion of hashing algorithms, exploring their effectiveness and relevance in identifying duplicate images.

In the literature of [Stallings, 2017], hash functions are described as mathematical algorithms that transform data of arbitrary length into a fixed-size string of bits, commonly referred to as hash values or hash codes. Hash functions are widely applied across various domains, including password storage and verification, digital signatures and authentication, hash tables and data structures, blockchain technologies and cryptocurrency, and file or data de duplication.

Despite the strength of hash functions, they are not without vulnerabilities and weaknesses. [Aradhana and Ghosh, 2021], has identified two prominent weaknesses: Collision Attacks and Birthday Attacks. A collision attack occurs when the hash function produces the same hash values, while a birthday attack involves the probability of finding a collision in the hash function. However, the scope of our research doesn't cover dealing with collisions as we are utilizing an already implemented and available library

This study focuses on utilizing hash functions to identify and eliminate duplicate images, thereby optimizing data storage efficiency. By comparing hash values, the system can quickly and effectively determine whether an image file already exists within a storage location or directory. This approach ensures a streamlined process for managing and storing digital image collections. Different hashing methods exist, but this study will utilize cryptographic and perceptual hashes.

2.2 Cryptographic Hashes

[Schneier, 2017], described that cryptographic hash functions are designed to produce a unique and consistent hash value for any given input, ensuring that even a minor change in the input results in a significantly different hash value. It is also noted in the scholarly article that cryptographic hashes are used to ensure data integrity, secure information, and enable efficient data retrieval.

There are several key characteristics of a good cryptographic hash function, and [Stallings, 2017], lists them as:

1. A cryptographic hash function must produce a deterministic output, meaning it must consistently produce the same hash value output for the same input data, ensuring reliable verification of data integrity.

2. A cryptographic hash function must be collision-resistant to ensure that it is computationally infeasible to find two different inputs that produce the same output.

3. It must also be computationally infeasible to determine the original input that is given the hash value.

4. A small change in input data should be able to produce a significantly different hash value, and this property is known as the Avalanche Effect.

There are various types of cryptographic hash functions, which will be discussed briefly.

2.2.1 MD5 - Message Digest Algorithm 5

The MD5 hash function was introduced by [Rivest, 1992], It takes an input message of arbitrary length and produces a unique 128-bit hash value. The internal process of MD5 involves processing the input message in 512-bit blocks through four iterative rounds of complex mathematical operations, which can be demonstrated from [Rivest, 1992]

1. Convert the image based on its corresponding size to bytes, e.g., an image of size 1MB is 1,048,576 bytes, since

   $$1\text{MB} = 1024 \times 1024 \text{ bytes}$$

   .

2. Calculate the original length in bits:

   $$1 \text{ byte} = 8 \text{ bits}$$
   $$\therefore 1,048,576 \text{ bytes} = 8,388,608 \text{ bits.}$$

3. **Padding:** The image is padded to a specific length divisible by 512 bits. This padding ensures all messages are processed in equal-sized chunks. It involves determining the number of '0' bits needed to make the total length congruent to 448 modulo 512. This is done using the expression:

   $$\text{Total Length} = L + 1 + k + 64 \equiv 448 \pmod{512}$$

   where L represents the original length in bits of the data, and k is the padding length, which can be computed using the expression:

   $$k = (448 - (L + 1) \bmod 512) \bmod 512$$

   - L: The original length of the message in bits. - k: The padding length — the number of '0' bits needed to make the length of the message congruent to 448 modulo 512.

   After the computation of k, k zero bits are appended to the bit stream, and finally, a 64-bit representation of the original message length L is appended to the end.

4. **Initialization:** This involves setting up initial hash values:

$$\begin{aligned}
A &= 0x67452301,\\
B &= 0xEFCDAB89,\\
C &= 0x98BADCFE,\\
D &= 0x10325476
\end{aligned}$$

5. **Processing in Rounds:** This involves processing each 512-bit block through four rounds of mathematical operations. It is implemented by dividing the padded image into 512-bit blocks, and for each block, dividing the block into sixteen 32-bit words $M_0, M_1, \ldots, M_{15}$.

   The four rounds of operations use the functions:

$$F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$$

   where $\wedge$ denotes the AND operation, defined as:

$$A \wedge B = \text{True if both } A \text{ and } B \text{ are true, otherwise False.}$$

$$G(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$$

   where $\vee$ denotes the OR operation, defined as:

$$A \vee B = \text{True if at least one of } A \text{ or } B \text{ is true, otherwise False.}$$

$$H(B, C, D) = B \oplus C \oplus D$$

   where $\oplus$ denotes the XOR (exclusive OR) operation, defined as:

$$A \oplus B = \text{True if } A \text{ and } B \text{ are different, otherwise False.}$$

$$I(B, C, D) = C \oplus (B \vee \neg D)$$

   where $\neg$ denotes the NOT operation, defined as:

$$\neg A = \text{True if } A \text{ is False, and False if } A \text{ is True.}$$

6. **Finalization:** After processing all message blocks, the final state is obtained.

2.2.2 SHA-256 Secure Hash Algorithm

According to [National Institute of Standards and Technology, 2015], SHA algorithms are iterative one-way hash functions that process a message to produce a condensed representation called a message digest. SHA-256 is more complex and secure than MD5, as it outputs a 256-bit hash, whereas MD5 outputs a 128-bit hash. Although SHA-256 operates similarly to the MD5 hash function, it employs different logical functions and constants in its processing rounds, making it resistant to the vulnerabilities that affect MD5. The logical functions used in SHA-256, as described in [National Institute of Standards and Technology, 2015], are:

- **Ch (Choose):** The **Ch** function takes three parameters E, F, and G, and is defined as the bitwise AND of E and F, XORed with the bitwise AND of the negation of E and G. This is expressed as:

$$\mathrm{Ch}(E, F, G) = (E \wedge F) \oplus (\neg E \wedge G)$$

- **Maj (Majority):** The **Maj** function takes three parameters A, B, and C, and is defined as the bitwise AND of A and B, XORed with the bitwise AND of A and C, XORed with the bitwise AND of B and C. This is expressed as:

$$\mathrm{Maj}(A, B, C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$$

- **Sigma Functions (Σ_0 and Σ_1):** The Σ_0 and Σ_1 functions involve bitwise rotations and shifts. These are used to manipulate the input bits in the processing rounds.

2.3 Perceptual Hashing

According to [Sharma, 2024] perceptual image hashing is a class of algorithms that produces content based image hashes that are excellent for identifying non-adversarial edits that happen when handling images on computer on a daily basis and when transfering files between computers. These edits may include changing file format, manipulating small features in an image, slightly reducing the size and quality of the file. A characteristic of perceptual hash functions is that in contrast to cryptographic hash a hash collision can occur when two perceptually distinct files are identified with the same hash. This is plausible since the goal of the hash function is to obtain comparable hashes from similar input.

Perceptual hashing provides a solution to find the similarity between images and there are different types of perceptual hashes according to [Viies, 2015] which is further discussed.

2.3.1 Average Hash (aHash)

Average hashing is one of the simplest ways to generate a hash of an image, which is invariant under content-preserving modifications. It involves resizing the image to a smaller size of 8 x 8 = 64 pixels before converting it to grayscale. The mean value of the 64 pixel/color values is then calculated. The binary hash string is computed depending on whether each pixel value is greater or less than the mean. A detailed mathematical representation of each step is described below.

1. The original image is resized to a smaller fixed dimension of 8 x 8 pixels. This step helps reduce complexity and focuses on the overall structure of the image. Mathematically, resizing the image is expressed as:

$$I_{\text{resize}}(i,j) = \frac{1}{h \cdot w} \sum_{x=0}^{h-1} \sum_{y=0}^{w-1} I\left(\frac{i \cdot h + x}{h}, \frac{j \cdot w + y}{w}\right)$$

   Here, $I(x,y)$ represents the original image, and $I_{\text{resize}}(i,j)$ represents the resized image. The variables h and w denote the height and width of the original image, respectively.

2. The resized image is then converted to grayscale to simplify the data to a single channel, focusing on luminance. This is done using the expression:

$$I_{\text{gray}}(i,j) = 0.3 \times R(i,j) + 0.59 \times G(i,j) + 0.11 \times B(i,j)$$

   In this case, $R(i,j)$, $G(i,j)$, and $B(i,j)$ are the red, green, and blue pixel values at coordinate (i,j) in the resized image.

3. The average pixel value of the grayscale image is computed using the mathematical expression for the 8 x 8 pixel values:

$$\text{average} = \frac{1}{N} \sum_{i=0}^{7} \sum_{j=0}^{7} I_{\text{gray}}(i,j)$$

Here, $N = 8 \times 8 = 64$ is the total number of pixels.

4. The binary hash values are computed by comparing each pixel value to the average using the mathematical expression:

$$H(i,j) = \begin{cases} 1, & \text{if } I_{\text{gray}}(i,j) > \text{average} \\ 0, & \text{otherwise} \end{cases}$$

   This results in a 64-bit binary string.

5. Finally, the binary hash values are converted to hexadecimal strings for easier comparison and storage.

2.3.2 Median Hash (mHash)

The Median hash is a perceptual hashing technique similar to the average perceptual hash, but it uses the median values of the pixel intensities instead of the mean to generate a hash. This approach can be more robust to certain types of image modifications, such as changes in brightness or contrast. The median hash is also resistant to noise and small random changes in pixel values. However, it is computationally complex as it requires sorting the pixel values. It is also sensitive to structural changes such as rotations or cropping.

The implementation of the median hash is similar to the average hash, but the main difference is in the calculation of the median pixel value instead of the average, which is done by sorting. The sorting is done by converting the 2D matrix into a 1D array represented by the expression:

$$\text{Pixels} = \{I_{\text{resize}}(0,0), I_{\text{resize}}(0,1), \ldots, I_{\text{resize}}(7,7)\}$$

This results in a list of 64 pixel values, which is then sorted in non-decreasing order. The median value is computed using the expression:

$$\text{mean} = \frac{p_{32} + p_{33}}{2}$$

2.3.3 Perceptual Hash (pHash)

The perceptual hash is driven by the discrete cosine transformation (DCT). The working principle of pHash involves first reducing the image to $32 \times 32 = 1024$ pixels and then converting it to grayscale using the same mathematical expression as the average hash. The image is then transformed using the DCT, represented by the expression:

$$D(u,v) = \frac{1}{4} \sum_{i=0}^{31} \sum_{j=0}^{31} I_{\text{gray}}(i,j) \cos\left(\frac{(2i+1)u\pi}{64}\right) \cos\left(\frac{(2j+1)v\pi}{64}\right)$$

Here, $D(u,v)$ represents the DCT coefficients, where u and v are the frequency indices in the horizontal and vertical directions, respectively, and

$$f'(x) = 2x\cos(x^2) - e^{-x}$$

$I_{\text{gray}}(i,j)$ is the grayscale value of the pixel at position (i,j).

The DCT hash is computed using the low DCT coefficients (i.e., the top $8 \times 8 = 64$ low-frequency components) in the list:

$$D_{\text{low}}(u,v) \quad \text{for } u,v \in \{0,1,\ldots,7\}$$

This involves selecting the low-frequency components where u and v are in the range from 0 to 7. The next step is to compute the median of these DCT coefficients.

The binary hash string is constructed based on whether each DCT frequency is lower or higher than the median, represented mathematically as:

$$H(u,v) = \begin{cases} 1, & \text{if } D_{\text{low}}(u,v) > \text{median} \\ 0, & \text{otherwise} \end{cases}$$

This results in a 64-bit binary string. pHash is suitable for its robustness to common image transformations such as scaling and rotations. However, it is computationally intensive.

2.3.4 Difference Hash (dHash)

dHash focuses on the visual structure by reducing the size of the image and eliminating higher frequencies. The hash string is computed based on the differences between neighboring columns. dHash shrinks the image to a smaller size ($9 \times 8 = 72$ pixels) and converts it to grayscale using a method similar to the average hash. The binary string is then calculated based on nearby pixels. The 9 pixels in each row generate 8 values for the differences between neighboring pixels, resulting in a total of 64 bits when the 8 rows are combined. This is mathematically represented by:

For each row in the grayscale image, compute the difference between each pair of adjacent pixels:

$$D(i,j) = I_{\text{gray}}(i,j+1) - I_{\text{gray}}(i,j)$$

Here, i ranges from 0 to 7 (for each row), and j ranges from 0 to 7 (for each column except the last one). $I_{\text{gray}}(i,j)$ represents the grayscale value of the pixel at position (i,j) in the image.

Create a binary hash by comparing each difference to zero:

$$H(i,j) = \begin{cases} 1, & \text{if } D(i,j) > 0 \\ 0, & \text{otherwise} \end{cases}$$

This results in 8 binary values per row, and with 8 rows, you get a total of 64 bits. $H(i,j)$ represents the binary hash value for each difference.

dHash is computationally efficient and robust to minor changes; however, it is sensitive to rotations and cropping and may miss important features of the image.

2.3.5 Wavelet Hash (wHash)

The wHash works on the principle of rescaling the image to a smaller size and converting it to grayscale using similar equations from the average hash. The hash string is then computed using the Discrete Wavelet Transformation (Haar transformation) to extract characteristics from the frequency domain. The equations for the transformation are represented as follows:

For the approximation (low-frequency) component $A(i,j)$, the formula is:

$$\begin{aligned} A(i,j) = \frac{1}{2} (&I_{\text{gray}}(2i,2j) + I_{\text{gray}}(2i+1,2j) \\ &+I_{\text{gray}}(2i,2j+1) + I_{\text{gray}}(2i+1,2j+1)) \end{aligned}$$

Here, $I_{\text{gray}}(i,j)$ represents the grayscale value of the pixel at position (i,j) in the image.

For the horizontal detail (high-frequency) component $H(i,j)$, the formula is:

$$\begin{aligned} H(i,j) = \frac{1}{2} (&I_{\text{gray}}(2i,2j) - I_{\text{gray}}(2i+1,2j) \\ &+I_{\text{gray}}(2i,2j+1) - I_{\text{gray}}(2i+1,2j+1)) \end{aligned}$$

For the vertical detail component $V(i,j)$, the formula is:

$$V(i,j) = \frac{1}{2} (I_{\text{gray}}(2i,2j) + I_{\text{gray}}(2i+1,2j)$$

$$-I_{\text{gray}}(2i, 2j+1) - I_{\text{gray}}(2i+1, 2j+1))$$

For the diagonal detail component $D(i,j)$, the formula is:

$$D(i,j) = \frac{1}{2}\left(I_{\text{gray}}(2i, 2j) - I_{\text{gray}}(2i+1, 2j)\right.$$
$$\left.-I_{\text{gray}}(2i, 2j+1) + I_{\text{gray}}(2i+1, 2j+1)\right)$$

Here, $A(i,j)$ is the approximation (low-frequency) component, while $H(i,j)$, $V(i,j)$, and $D(i,j)$ represent the horizontal, vertical, and diagonal detail (high-frequency) components, respectively.

It should be noted that the wHash does not include the lowest frequency portions, as they are excluded. The lowest frequency has only one data point that shows the image contrast.

2.3.6 Block Hash (bHash)

Block hash divides an RGB image into blocks, typically $16 \times 16 = 256$ blocks. If the image can be divided into the required number of blocks, a rapid version for hash computation is triggered. Otherwise, a slower version with extra processing is launched. We preprocess our images so that they can be divided into a number of blocks. After dividing the image into 256 blocks, the algorithm assigns a value to each block, which is the sum of all the RGB values in that block. The method then separates the 256 blocks into four groups of 64 blocks each. It computes the median of the block values for each group. Each bit in the binary string (256 bits) is calculated based on one block. The bit value is determined by whether the value of this block is greater than or less than the median of the block's associated group. The mathematical representation is as follows:

1. The original image is divided into $n \times n$ blocks, typically $16 \times 16 = 256$ blocks, using the expression:

   $$\left\lfloor \frac{W}{16} \right\rfloor \times \left\lfloor \frac{H}{16} \right\rfloor \text{ pixels}$$

   where $W \times H$ represents the image dimensions.

2. The block value is computed by summing all the RGB values in the block using the expression:

$$V_{i,j} = \sum_{x \in B_{i,j}} (R(x) + G(x) + B(x))$$

where $V_{i,j}$ is the value of block $B_{i,j}$, and $R(x), G(x), B(x)$ are the red, green, and blue pixel values at position x in the block.

3. The 256 blocks are then separated into 4 groups, each containing 64 blocks, by partitioning the blocks into quadrants and identifying each block group from the expression:

$$G_{i,j} = \begin{cases} 1, & \text{if } i < 8 \text{ and } j < 8 \\ 2, & \text{if } i < 8 \text{ and } j \geq 8 \\ 3, & \text{if } i \geq 8 \text{ and } j < 8 \\ 4, & \text{if } i \geq 8 \text{ and } j \geq 8 \end{cases}$$

where i and j are the block indices.

4. The median of each group is computed using the expression:

$$\text{median}(G_k) = \text{median}(\{V_{i,j} \mid B_{i,j} \in G_k\})$$

where $k = 1, 2, 3, 4$ represents the block group.

5. The binary hash is generated by creating a 256-bit binary string and comparing each block value with the median value of its group, calculated using the expression:

$$H(i,j) = \begin{cases} 1, & \text{if } V_{i,j} > \text{median}(G_k) \\ 0, & \text{otherwise} \end{cases}$$

where $H(i,j)$ represents the bit value for block $B_{i,j}$.

Block hash is highly robust to small changes and is relatively straightforward to implement or adapt by adjusting block size and the number of blocks. However, it is sensitive to geometric transformations, and its effectiveness is heavily dependent on how the image is divided into blocks.

2.4 Similarity Distance Measures

To evaluate the robustness and discriminative properties of image hashing, various distance and similarity metrics are required to cluster similar images effectively. Similarity is crucial in image duplicate detection because it provides the foundation for determining whether images are similar or entirely different. When hash values of images are generated from the hard drive or a local directory, it is necessary to measure the distance between these images. In other words, the computed hash values of two images should indicate how perceptually different they are. For this purpose, several metrics exist, which will be discussed.

2.4.1 Euclidean Distance

The research work by [Li and Lu, 2009] explains Euclidean distance as one of the common methods to measure similarities between images. This is done by converting the images into vectors based on the gray level of each pixel and then comparing the intensity differences between the pixels. By definition, an image with a fixed size $M \times N$ can be represented as a vector $x = \{x^1, x^2, \ldots, x^{MN}\}$ according to the gray levels of each pixel. The traditional Euclidean distance $d_E(x_1, x_2)$ between vectorized images x_1 and x_2 is defined as

$$d_E^2(x_1, x_2) = \sum_{k=1}^{MN} (x_1^k - x_2^k)^2 = (x_1 - x_2)^\top (x_1 - x_2)$$

It calculates the square root of the sum of the squared differences between two vector points. It is suitable for non-binary vectors like integers and it is perfect for measuring two hash values that is not represented in binary form.

2.4.2 Hamming Distance

According to [Hamming, 1950], the Hamming distance measures the similarity between two binary vectors by comparing them bit by bit and counting the number of differing bits. It measures the minimum number of substitutions required to change one string into another. By definition, the Hamming distance between two binary strings A and B is represented by:

$$HD = \sum |A_i - B_i|$$

where HD is the Hamming distance. If $HD = 0$, the hashes are identical. The higher the value, the more differences exist between the two hashes, indicating they are more distant from each other. However, the Hamming distance cannot exceed the length of the string.

The Hamming distance is the proposed metric for measuring similarity in this research. The operator $\oplus$ can be used to calculate the Hamming distance between two image hash values, given by:

$$HD(H_1, H_2) = \sum_{i=1}^{L} |h_1(i) \oplus h_2(i)|$$

where H_1 and H_2 are the binary hashes of two different images, both of length L. The terms $h_1(i)$ and $h_2(i)$ represent the i-th bit of H_1 and H_2, respectively. For valid comparison, both hashes must have the same length.

2.4.3 Normalized Hamming Distance

The Hamming distance can be normalized with respect to the string length L. The normalized Hamming distance is particularly useful for obtaining the ratio of differing elements, as stated in [Sharma, 2024]. It is given by:

$$NHD(H_1, H_2) = \frac{1}{L} \sum_{i=1}^{L} |h_1(i) \oplus h_2(i)|$$

where NHD is the normalized Hamming distance, H_1 and H_2 are binary hashes of two images of length L, and $h_1(i)$ and $h_2(i)$ represent the i-th bit of H_1 and H_2, respectively. For the Normalized Hamming distance, the result is expected to be close to 0 for similar images and close to 0.5 for dissimilar ones. The maximum value of the normalized hamming distance will be 1 but is very unlikely that it will ever reach it in case of image hashing, that is why the value close to 0.5 is more truthful for dissimilar images.

2.4.4 Manhattan Distance for Image Similarity

The Manhattan distance, also known as the city block distance or l_1 norm according to [Hanoun and Hashim, 2020] is the distance between two points measured along axesat right angles. It measures image similarity by summing the absolute differences between corresponding pixel intensities of two images. It is useful in image processing, where local variations in pixel values influence similarity assessment.

For two grayscale images I_1 and I_2 of size $M \times N$, represented as pixel matrices, the Manhattan distance is given by:

$$D_M(I_1, I_2) = \sum_{i=1}^{M} \sum_{j=1}^{N} |I_1(i,j) - I_2(i,j)|$$

where $I_1(i,j)$ and $I_2(i,j)$ represent the pixel intensity values at position (i,j) in the two images.

For feature vectors $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ extracted from images, the Manhattan distance can be expressed as:

$$D_M(X,Y) = \sum_{i=1}^{n} |x_i - y_i|$$

This distance metric is particularly effective in edge detection, texture analysis, and image retrieval, as it emphasizes pixel-wise variations. A smaller Manhattan distance indicates higher image similarity, whereas a larger distance signifies greater dissimilarity.

2.4.5 Chebyshev Distance for Image Similarity

According to [Deza and Deza, 2009], the Chebyshev distance, also known as the maximum metric or l_∞ norm, is a distance measure that considers the maximum absolute difference between corresponding components. The Chebyshev metric (also referred to as the lattice metric, chessboard metric, king-move metric, or 8-metric) is the L_∞-metric on $\mathbb{R}^2$, defined as:

$$\|x - y\|_\infty = \max\{|x_1 - y_1|, |x_2 - y_2|\}.$$

In the context of image processing, the Chebyshev distance serves as a measure of image similarity, particularly useful in applications where variations in individual pixel values must be accounted for.

For two grayscale images I_1 and I_2 of size $M \times N$, represented as pixel matrices, the Chebyshev distance is computed as:

$$D_C(I_1, I_2) = \max_{(i,j)} |I_1(i,j) - I_2(i,j)|,$$

where $I_1(i,j)$ and $I_2(i,j)$ denote the pixel intensity values at position (i,j) in the respective images.

For feature vectors $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ extracted from images, the Chebyshev distance is given by:

$$D_C(X,Y) = \max_{i=1}^{n} |x_i - y_i|.$$

This metric is particularly useful in texture analysis, pattern recognition, and template matching, where the largest deviation between pixel values determines similarity. A smaller Chebyshev distance indicates higher image similarity, while a larger value suggests greater dissimilarity.

2.4.6 Cosine Distance

The cosine of the angle between two vectors according to [Sinha and Shukla, 2013] indicates how closely they align in the same direction. This measure is commonly used in data mining to assess the cohesion within clusters. Given two vectors X = (x1, x2... xn) and Y = (y1, y2... yn), the cosine of the angle (cos) represents the degree of similarity between them in an n-dimensional space. It is a metric used to measure the dissimilarity between two non-zero vectors in an inner product space. In the context of image similarity, it is often used to compare the similarity between image feature vectors. The cosine distance is derived from cosine similarity, which measures the cosine of the angle between two vectors. It is defined as:

$$\text{Cosine Distance} = 1 - \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

where $\mathbf{A}$ and $\mathbf{B}$ are the image feature vectors, $\cdot$ denotes the dot product, and $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ represent the magnitudes (or norms) of the vectors. A cosine distance of 0 indicates that the vectors are identical in direction, while a value closer to 1 indicates greater dissimilarity. This metric is particularly useful in high-dimensional spaces, where the magnitude of the vectors may not be as informative as their orientation.

2.5 Technology and Tools for Implementation

The technologies and tools essential for implementing the image duplicate detection project will be explored in this section. A Python script will be developed, utilizing various libraries and packages within the Python ecosystem. Each tool plays a crucial role in different aspects of the project, and their functionalities and applications will be discussed in the following subsections.

2.5.1 SQL Database and SQL Alchemy

Structured Query Language (SQL) is a language used for interacting with databases. It allows us to create tables, and insert, delete, and search for records within a database. In this research, an SQL database will be used to store image hashes and metadata. The specific tool we will use is SQLAlchemy.

According to [Bayer, 2025], SQLAlchemy is a Python SQL toolkit and Object Relational Mapper (ORM) that provides application developers with the full power and flexibility of SQL. It offers several advantages over SQLite, including its ORM system, which allows for interaction with the database using Python objects instead of writing raw SQL queries. Additionally, SQLAlchemy provides database abstraction, supports complex queries, and manages relationships between tables efficiently. A more detailed explanation of how it works can be found in the documentation.

2.5.2 Pretty and Rich Functions

According to the documentation [McGugan, 2020], Pretty and Rich functions enhance the readability and presentation of console outputs in Python applications. In the image duplicate detection project, these functions will be used to print database information to the console for improved visualization. Detailed information of how it works can be found in the documentation.

2.5.3 Streamlit for GUI

According to [Streamlit Community, 2025], Streamlit is an open-source Python framework for creating interactive web applications. It enables developers to build and deploy data-driven applications quickly and easily. In the image duplicate detection project, Streamlit will be used to develop a front-end user interface, allowing users to interact effectively with the Python back-end script for detecting near-matching images.

3 Methodology

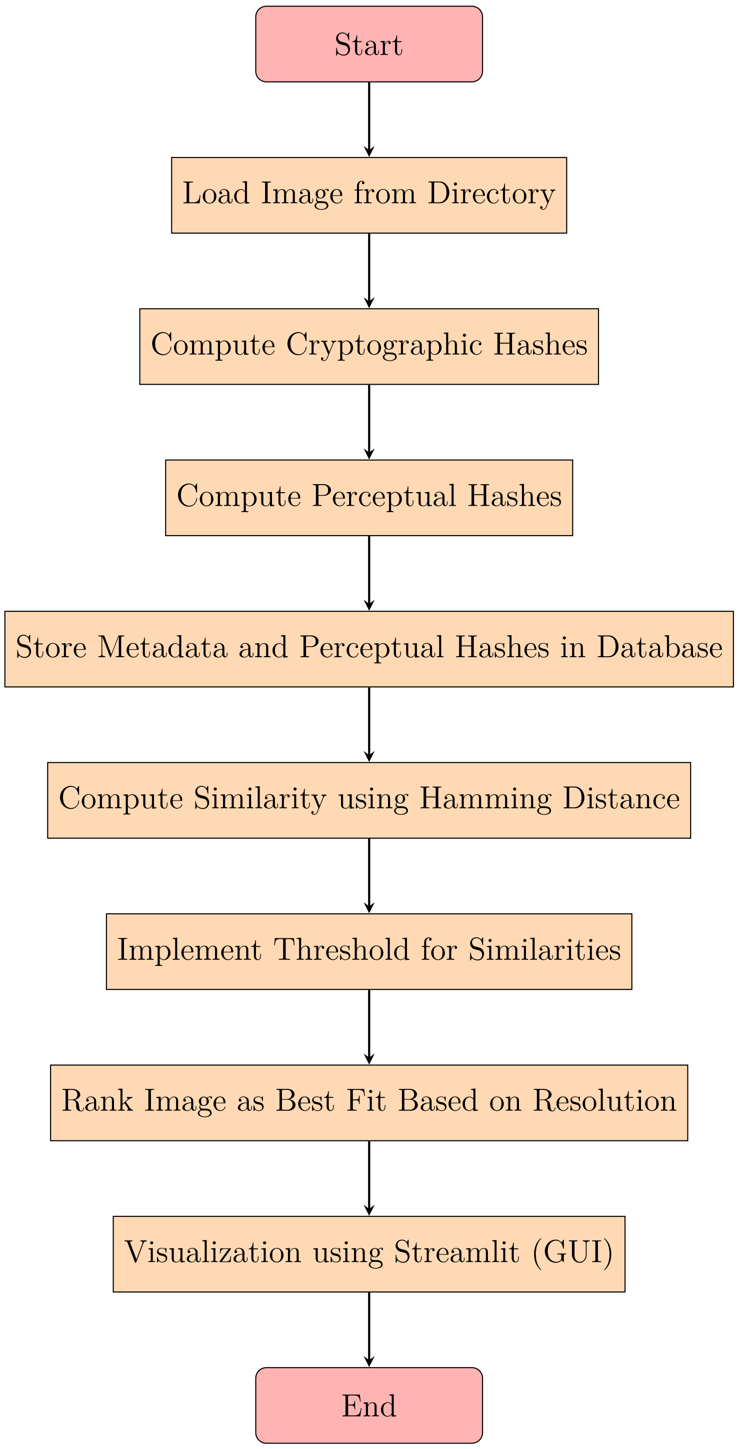
This chapter describes the framework and the main steps involved in the image duplicate detection utilizing the cryptographic hash as key to store the perceptual hashes and other meta data in database which is now used to compute similarities and the similar images are now ranked based on resolution.

The methodology begins with loading the image from drive and the generation of cryptographic hashes for each image. These hashes serve as unique identifiers, ensuring that each image can be distinctly recognized within the database. Alongside these cryptographic hashes, perceptual hashes are also generated. Unlike cryptographic hashes, perceptual hashes are designed to capture the visual essence of an image, allowing for the detection of images that are visually similar.

Once the hashes are generated, they are stored in a database along with relevant metadata, such as image resolution, format, and other attributes that may be useful for further analysis. This structured storage facilitates efficient retrieval and comparison of images.

The next step involves computing the similarity between images using their perceptual hashes using the hammings distance. By comparing these hashes, the system can identify images that are visually similar. This comparison is crucial for detecting duplicates, as it allows the system to recognize images that may have been altered slightly but still represent the same content.

After identifying similar images, the system ranks them based on their resolution. Higher resolution images are generally preferred, as they offer better quality and more detail. This ranking process ensures that when duplicates are detected, the best quality version of the image is prioritized. The proposed framework is implemented using python libraries.


Start
Load Image from Directory
Compute Cryptographic Hashes
Compute Perceptual Hashes
Store Metadata and Perceptual Hashes in Database
Compute Similarity using Hamming Distance
Implement Threshold for Similarities
Rank Image as Best Fit Based on Resolution
Visualization using Streamlit (GUI)
End

Workflow of Framework for Image Duplicate Detection

3.1 Load Images from Directory

The process of collecting images from a directory involves several key steps to ensure efficient and accurate image loading. Initially, the directory structure and file formats are defined, focusing on common image types such as PNG, JPG, JPEG, GIF, and BMP. These formats are widely supported and allow for versatile image processing.

Tools and libraries, such as **Python's os** for directory navigation and **Pillow** for image handling, are employed to streamline the collection process. These libraries provide robust methods for accessing and verifying images, ensuring that only valid files are processed.

The system then iterates through the directory, identifying files with supported formats. Each image is opened and verified to ensure it is not corrupted. Successfully loaded images are stored, while any errors encountered during loading are logged for review.

3.2 Compute Cryptographic Hashes

The choice of cryptographic hash functions, such as SHA-256, is crucial for ensuring the integrity and uniqueness of image identifiers. SHA-256 is a widely used cryptographic hash function that produces a fixed-size, 256-bit hash value. It is known for its strong collision resistance, meaning it is computationally infeasible for two different inputs to produce the same hash. This property makes SHA-256 an ideal choice for generating unique identifiers for images, ensuring that each image can be distinctly recognized and verified within a database or system.

The process of computing cryptographic hashes for each image involves reading the image file in binary mode and processing it in chunks. This method is efficient and allows for handling large files without consuming excessive memory. The Python hashlib library provides a straightforward way to implement this process. By iterating over the image file in chunks, the SHA-256 hash is updated incrementally until the entire file is processed. The final hash value is then converted to a hexadecimal string, which serves as a unique identifier for the image. This approach ensures that the hash computation is both efficient and reliable.

Using cryptographic hashes offers several benefits and in our case the hashes serve as unique identifiers, facilitating efficient storage and retrieval of images in databases. This is especially useful in applications such as duplicate detection, where the hash can quickly determine if an image has already been processed. Overall, cryptographic hashes enhance the robustness and reliability of image management systems.

3.3 Computing Perceptual Hashes Using aHash and pHash

Perceptual hashing, as discussed in Chapter 2, involves applying Discrete Cosine Transform (DCT) to extract frequency components, which are then used to generate a binary hash by comparing them with average values. On the other hand, the Average Hash (aHash) is a simpler yet effective method for computing a perceptual hash of an image. It operates by resizing the image to a smaller, fixed size, converting it to grayscale, and creating a binary hash based on the average pixel value. Below is a step-by-step explanation using a matrix example:

1. **Convert the Image to Grayscale:** Suppose we have an 8×8 grayscale image represented by the following matrix, where each number denotes a pixel's intensity (ranging from 0 to 255):

$$\begin{bmatrix} 100 & 102 & 103 & 105 & 110 & 112 & 115 & 117 \\ 98 & 100 & 102 & 104 & 108 & 110 & 113 & 115 \\ 97 & 99 & 101 & 103 & 107 & 109 & 112 & 114 \\ 96 & 98 & 100 & 102 & 106 & 108 & 111 & 113 \\ 95 & 97 & 99 & 101 & 105 & 107 & 110 & 112 \\ 94 & 96 & 98 & 100 & 104 & 106 & 109 & 111 \\ 93 & 95 & 97 & 99 & 103 & 105 & 108 & 110 \\ 92 & 94 & 96 & 98 & 102 & 104 & 107 & 109 \end{bmatrix}$$

2. **Compute the Average Pixel Value:** The average pixel intensity is calculated as:

$$\text{Average} = \frac{\sum \text{all pixel values}}{64} = 104$$

3. **Generate the Binary Hash:** Each pixel value is compared to the average intensity. If a pixel is greater than or equal to the average, it is assigned a 1; otherwise, it is assigned a 0, forming the following binary matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

4. **Convert the Binary Matrix to a Hash:** The binary matrix is flattened into a single binary string, which can then be represented in hexadecimal or decimal form as the final hash.

McKeown and Buchanan [2023] investigated the Hamming distance distributions of different perceptual hash methods. His findings indicate that aHash produces the smallest Hamming distances between original and modified images, implying that it is less robust to changes. In contrast, pHash, which is based on the DCT, demonstrated higher robustness to modifications such as scaling and compression, exhibiting a more normal distribution in Hamming distances.

3.3.1 Geometric Transformations for Robustness Evaluation

To assess the robustness of aHash, three variations of the original images were generated through geometric transformations:

- A 15-degree clockwise rotation.
- A 15-degree counterclockwise rotation.
- A shear transformation with a 5% shift in both the X and Y directions.

The rotation transformation is expressed as:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For this experiment, $\theta = 15^\circ$ for both clockwise and counterclockwise rotations.

The shear transformation is represented by the following matrix:

$$S = \begin{bmatrix} 1 & \text{shear}_x & 0 \\ \text{shear}_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In our case, the shear values are 0.05 for both X and Y directions, yielding:

$$S = \begin{bmatrix} 1 & 0.05 & 0 \\ 0.05 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These transformations simulate potential real-world modifications that images might undergo. While additional variations could be explored to further evaluate robustness, these transformations were selected for simplicity and computational efficiency.

Based on the findings, aHash was selected for this project due to its rapid computation and efficient Hamming distance calculations.

3.4 Compute Similarities

In this section, we introduce a method to identify similarities between images using distance measures. The proposed approach involves computing the Hamming Distance between the average hashes (aHashes) of two images. Initially, clustering techniques such as K-Nearest Neighbors or similar algorithms were considered. However, due to the challenge of defining a feature space for clustering, the Hamming Distance was determined to be the most suitable metric. It quantifies the number of differing bits between two binary strings, providing a straightforward way to measure similarity.

Suppose we have two aHash values for two images:

$$\text{aHash}_1 = 10101010$$

$$\text{aHash}_2 = 10011011$$

To compute the Hamming Distance, we compare each corresponding bit in the two hashes:

Position	aHash_1	aHash_2
1	1	1
2	0	0
3	1	0
4	0	1
5	1	1
6	0	1
7	1	1
8	0	1

The differing bits are at positions 3, 4, 6, and 8. Therefore, the Hamming Distance is 4.

By calculating the Hamming Distance, we can determine how similar two images are. A smaller distance indicates greater similarity, while a larger distance suggests more differences. This method is efficient and effective for quickly comparing large sets of images, making it a valuable tool in image processing applications.

The threshold for determining similarity in a database of images will be user-defined and based on the maximum pairwise Hamming Distance among all images in the database. This threshold allows for flexibility, enabling users to adjust the sensitivity of similarity detection according to their specific needs. By setting an appropriate threshold, users can effectively filter out images that are too dissimilar, ensuring that only closely related images are considered similar. Once similar images are identified, they are ranked based on resolution, with higher resolution images prioritized as they offer better quality and detail. This approach not only enhances the accuracy of duplicate detection but also ensures that the best quality images are retained, making it a robust solution for image management and processing applications.

3.5 Storing Metadata in a Database Using SQLAlchemy and Pretty Functions

The database schema for storing image metadata is designed to efficiently manage and retrieve information related to each image. This schema is implemented using SQLAlchemy's declarative base, which provides a structured and intuitive way to define database tables. Each image entry consists of multiple attributes, including unique identifiers, file paths, timestamps, and perceptual hashes corresponding to modified versions of the image. The schema enforces data integrity through appropriate constraints and data types, ensuring consistency and optimized query performance.

SQLAlchemy is utilized for database interactions, serving as a powerful and flexible Object-Relational Mapping (ORM) framework. It abstracts raw SQL queries, allowing developers to interact with the database using Python classes and objects. This enhances maintainability and scalability while reducing the risk of SQL injection attacks. The `initialize_db` function is responsible for setting up the SQLite database, creating necessary tables based on the defined schema. Additionally, SQLAlchemy's session management ensures thread-safe interactions, enabling efficient transaction handling and rollback mechanisms when needed. These features collectively enhance the reliability and robustness of database operations.

In addition to database management, Pretty Functions play a crucial role in structuring and processing image metadata. These functions are instrumental in formatting data, generating modified image variations, and computing their perceptual hashes. By ensuring that the stored metadata remains well-organized and easily accessible, Pretty Functions streamline data retrieval and manipulation processes. Their integration within the image processing workflow significantly improves overall efficiency, making it easier to analyze, compare, and manage large datasets of images.

The combination of SQLAlchemy's ORM capabilities and the utility of Pretty Functions ensures a scalable and efficient approach to metadata storage. This approach not only simplifies interaction with the database but also optimizes performance when dealing with extensive image datasets.

3.6 GUI for Visualization Using Streamlit

The Streamlit application is designed to facilitate image processing and similarity detection through an intuitive graphical user interface (GUI). The application consists of two main functionalities: processing a folder of images and uploading a single image to find similar images based on perceptual hashing.

- **Process Folder Section:** Users can input the path to a directory containing multiple images. Upon clicking the `Process Folder` button, the application verifies whether the directory exists. If valid, the IMDEDU class initializes the directory path and loads images, displaying the total number of successfully loaded images.

  To efficiently manage image metadata, SQLAlchemy is used for database interactions, leveraging an SQLite database. The application recreates the table schema and stores metadata, including perceptual hashes of processed images. This structured approach ensures that image data is well-organized and readily accessible for similarity comparisons.

- **Upload Image to Find Similarity Section:** Users can upload an image file, which is then displayed and temporarily saved for processing. The application computes the Hamming distances between the uploaded image's perceptual hash and those stored in the database.

  Users can adjust a threshold slider to filter similar images based on their Hamming distance. The results are displayed in a table, showing matching images along with their computed distances. Additionally, the best-matching image based on resolution is highlighted.

  For further analysis, users can download the filtered results as an Excel file. This section provides an intuitive tool for visualizing and analyzing image similarities, enhancing the user's ability to manage image datasets effectively.

In summary, the methodology outlined in this chapter provides a comprehensive framework for image duplicate detection, leveraging cryptographic and perceptual hashing techniques. The integration of SQLAlchemy for database management and Streamlit for visualization ensures efficient processing and user-friendly interaction with image data. By implementing robust similarity measures and ranking mechanisms, the system effectively identifies and prioritizes high-quality images. The results of this methodology, including its effectiveness and potential improvements, will be presented and discussed in the next chapter.

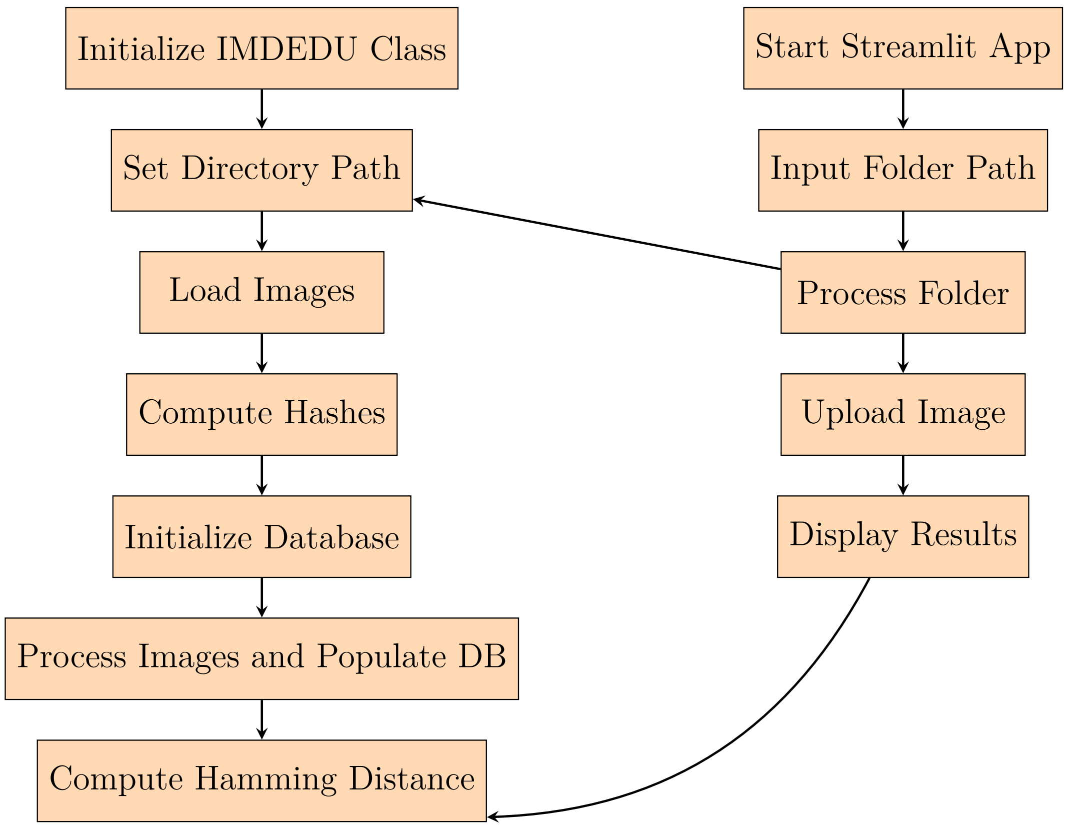
4 Results and Discussion

This chapter presents and analyzes the results obtained from the proposed method for identifying duplicate images. The discussion highlights the effectiveness of the approach, its strengths, and potential areas for improvement. A detailed analysis of the results provides insights into the accuracy and efficiency of the duplicate detection process. Additionally, these findings are considered in the context of real-world applications, offering a comprehensive understanding of the project.

The implementation consists of two main scripts: a backend , `main.py`, and a frontend script, `app.py`, which serves as the graphical user interface (GUI). The `main.py` script handles core image processing tasks, including loading images from a specified directory, computing cryptographic and perceptual hashes, and managing image metadata within a database using SQLAlchemy. It efficiently processes image transformations and calculates Hamming distances to identify duplicates.

The images obtained could be from a specified directory on an external hard drive or with the computer's local drive, the path to which is manually inputted into the program. Conversely, the `app.py` script utilizes Streamlit to create an interactive user interface that seamlessly connects with the backend. Users can specify folder paths, process images, and upload individual images for similarity detection. The GUI displays results in an intuitive manner, providing real-time feedback and allowing users to download filtered data. Together, these scripts form a cohesive system that integrates robust backend processing with an accessible frontend interface, enabling efficient image management and duplicate detection. The following flowchart illustrates the step-by-step workflow of the duplicate image detection system, showcasing the interactions between the backend and frontend components.

Figure 4.1: Flowchart Representation of the Image Processing System



4.1 Image Collection Analysis

A sample set of approximately 38 images was collected from various sources on a personal computer. These images encompass a diverse range of features, including portraits of people, images of wall cracks, and several other distinct characteristics. The analysis was conducted on this relatively small dataset to evaluate the effectiveness of the proposed method. Despite the limited size, the dataset was chosen to represent a variety of real-world scenarios that the system might encounter. The diversity of the sample images ensures that the implementation is tested against different types of visual content, which is crucial for assessing its reliability in practical applications.

4.2 Cryptographic Hash Results

The cryptographic hashes for all images were computed using the SHA-256 algorithm, facilitated by Python's built-in 'hashlib' library. Each resulting hash is a 64-character long hexadecimal string, serving as a unique identifier for each image. This hash is used as a key for storing image metadata and perceptual hashes in the database. The cryptographic hash is particularly valuable because even a slight alteration in any property of an image results in a completely different hash value. This characteristic is crucial for distinguishing between identical and modified images, ensuring data integrity and accurate duplicate detection.

4.3 Perceptual Hash Results

The perceptual hashes of the original image and its three variations, as described in Chapter 3, were computed using the aHash method. This was implemented with the 'imagehash' library in Python. The resulting hash values are 16 characters long. This hashing method was selected due to its computational efficiency, which is crucial when processing large numbers of images, as discussed in Chapter 3. However, it has been noted by McKeown and Buchanan [2023] that pHash is more robust to slight alterations compared to aHash.

The perceptual hash provides a compact representation of the image's visual content, allowing for efficient comparison and duplicate detection. While aHash offers speed and simplicity, pHash's robustness makes it more suitable for applications where images may undergo minor modifications.

The resulting cryptographic and perceptual hashes are illustrated in the figure below, highlighting the differences and applications of each method.

Figure 4.2: Sample cryptographic and perceptual hash results.

```
■[1m{■[0m
    ■[32m'1f1a20e54273cf20d5342de29e7591b829cede977040303bc8cd73aac871212d'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'3f3b380d1e3c6400'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'1e7efeffff7f7e70'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'787f7ffffffe7e0
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'3f3bba3e1efefe00'■[0m
    ■[1m}■[0m,
    ■[32m'0046be088fdafdb8d921d937ce11973058b21d8f26a974d4009d6d267075fb44'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'0040f8fcfeffffff'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'0e7efeffff7f7e78'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'707e7ffffffe7e1
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'00fefefefefefe80'■[0m
    ■[1m}■[0m,
    ■[32m'b36aa0e46beed02075688769da64b35b73fbf36a6a2d46eaafe86244846461d6'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'3f3f3f7736220000'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'1e7efeffff7f7650'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'787e7fffffbe6e0
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'ffbfbffefee62e00'■[0m
    ■[1m}■[0m,
    ■[32m'ed93c56ca71f42a1401d28f99e83cd74e54f50443f2c3c182186c5763f4727ee'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'00083f1f071f3f0f'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'1cfe7eff3f7f7f38'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'385f7f3ffffefe1
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'1e3abe3e1ede7e00'■[0m
    ■[1m}■[0m,
    ■[32m'61bce4170da81925367811ea80a38cbaf19bca71abad2a2ea07790662fb8048f'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'0000446ceefcfcfc'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'0a6efeffff7f7e78'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'70767dfffffe7e1
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'20644efefefcfee0'■[0m
    ■[1m}■[0m,
    ■[32m'ae2a1aefeb8fa1b355096cd1105e818a28cfe5aa57180a4ae927c15c5a9988d4'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'002078307cfefcfc'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'0e7efeffff7f7e78'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'707e7ffffffe7e1
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'00f87efcfefefec0'■[0m
    ■[1m}■[0m,
    ■[32m'f7c79527058e6d61e7302b26ac0cc048f02d3b5ab98d9446f9b08c0df25ac6c8'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'e0f4fefcf4f02000'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'1cfefefeff7f7f38'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'387f7ffffefefe1
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'fafefefefefaea00'■[0m
    ■[1m}■[0m,
    ■[32m'd36c9666cd237133b1a31b9cc7a86169c5805842633bc8b2e091fa7d9ae9431e'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'3f3f7f3e0c1e0400'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_Clockwise'■[0m: ■[32m'1e7efeffff7f7e70'■[0m,
        ■[32m'Perceptual_Hash_rotation_15degrees_CounterClockwise'■[0m: ■[32m'787e7ffffffe7e0
        ■[32m'Perceptual_Hash_Shear_Low_Difference'■[0m: ■[32m'7f7f7e3e7e7e3e00'■[0m
    ■[1m}■[0m,
    ■[32m'c8ebfb7ac686374089811da10a67d32d85059aff458d09aa8b904e50f9c14386'■[0m: ■[1m{■[0m
        ■[32m'Perceptual_Hash'■[0m: ■[32m'00040f2f3f7f2f07'■[0m,
```

4.4 Similarity Computation Outcomes

The Hamming Distance, as discussed in Chapter Three, was used to compute the similarities between images by calculating pairwise distances. Suppose we have Image A (test image) and Image B (every other image in the dataset), each consisting of a set of 4 perceptual hashes. The pairwise Hamming distance is computed between every perceptual hash of set A and set B, and the minimum distance is reported as the minimum Hamming distance or similarity between the two images.

Mathematically, let

$$A = \{a_1, a_2, a_3, a_4\}$$

and

$$B = \{b_1, b_2, b_3, b_4\}$$

represent the sets of perceptual hashes for Images A and B, respectively. The pairwise Hamming distances are calculated as:

$$d(a_i, b_j) = \text{Hamming}(a_i, b_j) \quad \text{for } i, j \in \{1, 2, 3, 4\}$$

The minimum Hamming distance is then given by:

$$\text{min\_distance} = \min\{d(a_i, b_j) \mid i, j \in \{1, 2, 3, 4\}\}$$

This minimum distance represents the similarity between the two images. A similarity matrix is then constructed by comparing every image with every other image in the database to assess their similarities. This matrix is illustrated in the figure below.

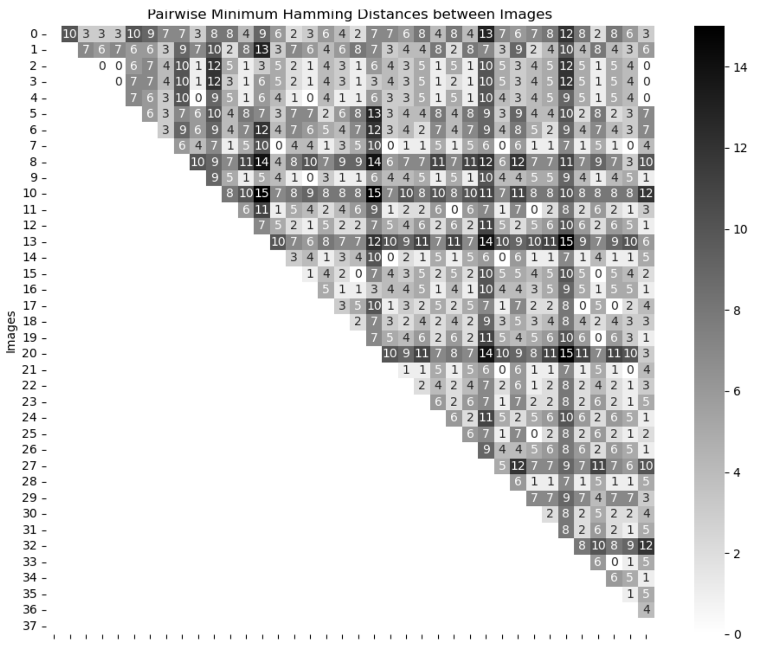


Figure 4.3: Similarity Matrix Plot

Since the similarity matrix is symmetrical, only the upper triangular matrix was computed, and all diagonals have a similarity of 0. Based on the maximum and minimum Hamming distances, a threshold is implemented, allowing users to adjust the extent to which similarities are measured. This threshold is adjustable according to the user's preference, providing flexibility in determining the level of similarity that is considered significant. This approach improves the usability of the system, allowing users to tailor the similarity detection process to their specific needs.

4.5 Database Storage Efficiency

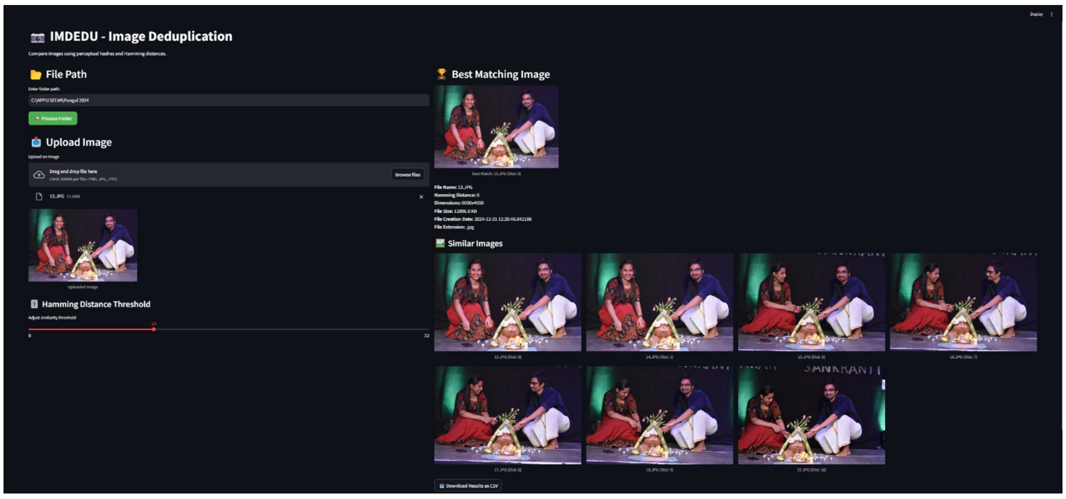
The metadata of the images was efficiently stored in an ORM database using SQLAlchemy, which is crucial for organizing and retrieving information without the need to repeatedly load the actual image files. This is particularly beneficial in large databases, where the cryptographic hash of each image serves as the key to the database.

The performance of the database was found to be highly efficient, enabling rapid queries and facilitating the extension of features through predefined methods. This approach ensures that the system can scale effectively, maintaining quick access to image metadata and supporting additional functionalities as needed.

4.6 Visualization Insights

The GUI developed with Streamlit for the IMDEDU is designed to provide an intuitive and efficient user experience. The interface features a sleek, dark-themed layout that highlights key functionalities. Users can easily input the file path to your hard drive or external storage drive connected to the computer to process a folder of images and upload individual search images you want to find the duplicate through a drag-and-drop feature. The GUI displays the best-matching image by resolution alongside other similar images in the folder, providing immediate visual feedback. Key details such as file name, Hamming distance, dimensions, and file size are clearly presented, allowing users to quickly assess the similarity results.

The simplicity and clarity of the interface make the user experience positive. The use of clear icons and labels enhances usability, making it accessible even to those with limited technical experience. The ability to adjust the similarity threshold offers flexibility, catering to diverse user needs. Overall, the GUI effectively balances functionality and user-friendliness, making it a valuable tool for image comparison tasks.


IMDEDU - Image Deduplication
Compare images using perceptual hashes and Hamming distances.
File Path
Enter folder path:
Process Folder
Upload Image
Upload an image
Drag and drop file here
Limit 200MB per file • PNG, JPG, JPEG
Browse files
Uploaded Image
Hamming Distance Threshold
Adjust similarity threshold
0
32
Best Matching Image
Similar Images
Download Results as CSV

C:\APPU SELVA\Pongal 2024

13.JPG 12.6MB

10

Best Match: 13.JPG (Dist: 0)

File Name: 13.JPG
Hamming Distance: 0
Dimensions: 6000x4000
File Size: 12896.0 KB
File Creation Date: 2024-12-21 12:20:46.042108
File Extension: .jpg

13.JPG (Dist: 0) | 14.JPG (Dist: 1) | 15.JPG (Dist: 6) | 16.JPG (Dist: 7)
17.JPG (Dist: 6) | 18.JPG (Dist: 9) | 19.JPG (Dist: 10)

Figure 4.4: GUI Interface

5 Conclusions and Recommendations

5.1 Summary of Findings

The research aimed to develop a system for efficiently detecting similarities and duplicates between images using hashing techniques, including cryptographic hashes (SHA-256), perceptual hashes (average hash), and Hamming distances. The main findings of the research are as follows:

1. Cryptographic hashes proved effective in ensuring the integrity of images, allowing for differentiation even among similar images. This facilitated efficient storage and retrieval of image information in the database.

2. Perceptual hashes provided a robust method for identifying similarities between images, even when alterations such as resizing, color changes, and rotations were present.

3. Hamming distances served as a valuable metric for measuring and clustering similarities between images, which was visualized using a similarity matrix.

4. The SQLAlchemy database demonstrated efficiency in storing image metadata, including hash values, enabling faster retrieval and querying without the need to load full images.

5. Streamlit was instrumental in providing a user-friendly GUI interface, allowing end users to interact seamlessly with the system.

The results of the study indicate that the proposed techniques using hash functions are highly effective in detecting multiple versions of images scattered across various storage locations, such as computers and external drives. This approach addresses the need for efficient organization, archiving, and retrieval of similar images, with applications in various fields. The system's ability to manage and process large datasets of images highlights its potential for widespread use in diverse fields.

5.2 Contributions

This research supports and expands upon existing knowledge in the field of Image Duplicate Detection. Previous studies have demonstrated that hashes are valuable for ensuring image integrity and enhancing image search systems. This study further validates these findings by providing a practical application where both cryptographic and perceptual hashes are used together to optimize the management of large image datasets. By demonstrating how perceptual hashes can effectively detect visual similarities, this research highlights their utility in identifying duplicates even when images have undergone alterations.

The integration of cryptographic hashes ensures that each image is uniquely identifiable, facilitating efficient storage and retrieval. Meanwhile, perceptual hashes allow for the recognition of visually similar images, offering a robust solution for duplicate detection. This dual approach not only enhances existing methods of image categorization but also provides a more nuanced strategy for image analysis. The study's findings contribute to the development of more sophisticated image management systems, with potential applications in various fields such as digital archiving, social media, and digital forensics. By bridging the gap between theoretical research and practical implementation, this work offers significant advancements in the field of image processing.

5.3 Limitations

Several limitations were encountered during the research process:

1. **Data Quality:** The image dataset used for testing varied in quality, including low-resolution and compressed images, which affected the accuracy of some hash values. This variability may have resulted in perceptual hashes that were less accurate for low-resolution or highly compressed images.

2. **Algorithm Complexity:** While average hashing provides a good balance between speed and performance, it may not always detect subtle visual differences, such as slight changes in color saturation or lighting. Although aHash was effective for this study, it may not be the best choice for more complex images or datasets involving significant transformations. Other perceptual hashing methods, like pHash, could be considered for greater accuracy.

3. **Threshold Selection:** The threshold value used to determine image similarity (i.e., similarity score) was chosen arbitrarily based on user-

defined perspectives. This could lead to false positives or false negatives when identifying visually similar images. More robust techniques could be explored to accurately cluster similar images together.

Despite these limitations, the study provides valuable insights into how hashing techniques can be applied to image databases and storage systems. The results hold significant value for many real-world applications, offering a foundation for further research and development in image processing.

5.4 Future Works

While this study successfully demonstrated the use of cryptographic and perceptual hashing techniques for image analysis, several areas can be explored further:

1. Research can be expanded to include more sophisticated perceptual hashing algorithms, such as pHash, to improve accuracy. Additionally, exploring machine learning-based hashing techniques could offer greater flexibility and adaptability in identifying similarities in complex images.
2. Future work could also involve testing methods on larger and more diverse datasets, including images from different sources such as social media, scientific data, and e-commerce websites. This would help further validate the approach and enhance its applicability.

Bibliography

Aradhana and D. S. M. Ghosh. Review Paper on Secure Hash Algorithm With Its Variants. *International Journal of Technical Innovation in Modern Engineering & Science*, 3(5):01–07, Nov. 2021.

M. Bayer. SQLAlchemy - The Database Toolkit for Python, 2025. URL `https://www.sqlalchemy.org/`. Version 2.0, Accessed: 2025-03-04.

M. M. Deza and E. Deza. *Encyclopedia of Distances*. Springer, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-642-00234-2.

R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

T. Hanoun and K. Hashim. Modify manhattan distance for image similarity. *Open Journal of Science and Technology*, 2:12–16, 02 2020. doi: 10.31580/ojst.v2i4.984.

J. Li and B.-L. Lu. An Adaptive Image Euclidean Distance. *Pattern Recognition*, 42(3):349–357, 2009. doi: 10.1016/j.patcog.2008.07.017.

W. McGugan. Rich: Python Library for Rich Text and Beautiful Formatting in the Terminal, 2020. URL `https://github.com/Textualize/rich`.

S. McKeown and W. J. Buchanan. Hamming distributions of popular perceptual hashing techniques. *Forensic Science International: Digital Investigation*, 44:301509, 2023. doi: 10.1016/j.fsidi.2023.301509. Selected papers of the Tenth Annual DFRWS EU Conference.

National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS PUB 180-4, National Institute of Standards and Technology, August 2015.

R. L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley, 2017.

S. Sharma. Distance distributions and runtime analysis of perceptual hashing algorithms. *Journal of Visual Communication and Image Representation*, 104:104310, 2024. doi: 10.1016/j.jvcir.2024.104310.

A. K. Sinha and K. Shukla. A Study of Distance Metrics in Histogram Based Image Retrieval. *International Journal of Computers & Technology*, 4(3): 821–830, 2013.

W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson, 7th Global Edition edition, 2017.

Streamlit Community. Streamlit: Turn Python Scripts into Beautiful Web Apps, 2025. URL `https://docs.streamlit.io/get-started`. Version 1.42.0, Accessed: 2025-02-05.

K. K. Thyagharajan and G. Kalaiarasi. A review on near-duplicate detection of images using computer vision techniques. *Archives of Computational Methods in Engineering*, 28(3):897–916, May 2021. doi: 10.1007/s11831-020-09400-w.

V. Viies. Possible Application of Perceptual Image Hashing. *Tallinn University of Technology, Faculty of Information Technology Department of Computer Engineering, Master Thesis*, 2015.