Yet Another *Trie* to Hash Image

Changdae Son
Dept. of Electrical and Computer Engineering
Seoul National University
changdae20@snu.ac.kr

Abstract

In this paper, we propose an algorithm for efficient image retrieval in large-scale databases. Our algorithm leverages a unique approach by quantizing feature vectors into binary values, resembling hash values, and efficiently utilizes the "Trie" data structure for retrieval. This allows for quick retrieval times with an average complexity of $\mathcal{O}(\#$ of Features$)$. *Unlike existing deep hashing algorithms or feature extractors, our algorithm does not require additional learning processes when the database is updated. Experimental results using various feature sets, including low-level features based on RGB pixels and high-level features derived from corners and blobs, demonstrate the robustness and efficiency of our algorithm. Our approach has the potential for practical applications in image retrieval tasks. Source code is available at https://github.com/changdae20/Yet-Another-Trie-To-Hash-Image.*

1. Introduction

1.1. Image Retrieval Task

Image retrieval is a fundamental challenge in computer vision, where the task is to efficiently search for similar images within a large-scale database.[1] Most existing algorithms follow a two-step approach, where they first extract features from an image and then compare the feature vector of the input image with those stored in the database.

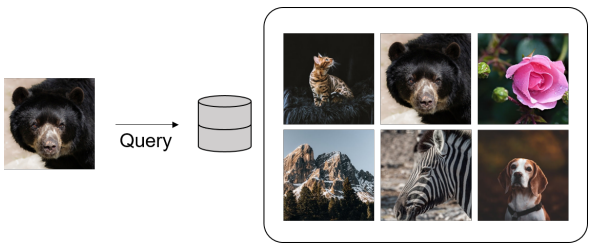

Figure 1. Query image (left) is given to a database(right)

In this paper, we address the challenge of efficient image retrieval, particularly focusing on photos taken with smartphones or cameras. By narrowing down our research to images captured by these devices, we aim to tackle the specific variations that commonly occur in such images. These variations include translation, rotation, projection, and hue shift, shown as Figure 2, which are inherent to the image capture process.

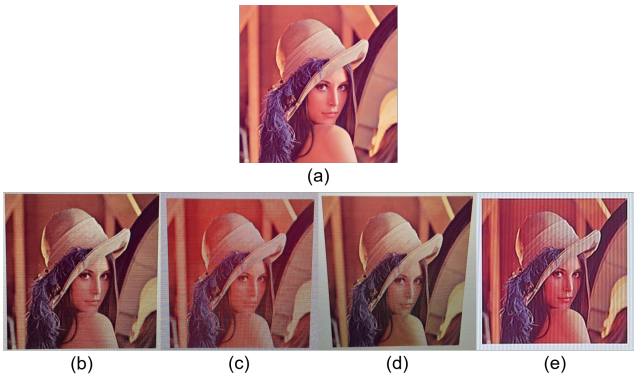

Figure 2. Original image (a) and its possible variations : (b), (c), (d), and (e).

1.2. Related Work on Image Retrieval Algorithms

The field of image retrieval has witnessed numerous advancements, with various feature extracting and algorithms proposed to improve the efficiency and accuracy of the naïve nearest neighbor search. One approach to improve nearest neighbor search algorithm, is to reduce operations by bounding the target group to be compared[2]. And recently, there have been many attempts to improve feature quality of image using deep neural net, such as Transformer[3], Attention[1], and Autoencoder[4], which excel at capturing high-level semantic information. However, a significant drawback of deep neural networks is the need for retraining the feature extractor when the database is updated, leading to substantial time overhead. On the other hand, our proposed algorithm does not only require additional learning processes but also

less affected by the size of the database.

1.3. Initial Attempt : Hash Function and Binary Search

In our initial exploration, we attempted to tackle the image retrieval task using a hash function with hexadecimal representation as follows.

$$\{x \in \mathbb{R} | 0 \leq x \leq 1\}^{W*H} \mapsto \mathbb{Z}_{16}^{F}$$

Once the feature has been extracted as a hexadecimal string, we can utilize a binary search algorithm to find the hashed value in the database. However, we encountered a significant limitation with binary search, as it necessitated "exact equality" of hash values for matching, making it an inefficient algorithm for practical applications. Recognizing this restriction, we concluded that quantizing the feature vector into a binary vector representation could provide a more flexible solution.

1.4. Proposal : Hashing with Trie

To address the limitations of the initial approach, we propose a novel algorithm that quantizes the feature vector into a binary vector, resembling a hash value. Subsequently, we utilize a *Trie* data structure to efficiently insert and find these quantized feature vectors. Our algorithm offers the advantage of fast retrieval times, with an average time complexity of $\mathcal{O}(\log N)$, while guaranteeing a worst-case seek time complexity of $\mathcal{O}(N)$. To account for variations near the threshold of 0.5 in the binary vector representation, we introduce a margin, denoted as ξ, allowing values in the vicinity of 0.5 to match both 0 and 1. Furthermore, we employ a *Trie* data structure and queue-based traversal using the breadth-first search (BFS) strategy to efficiently search the quantized feature vectors.

2. Our Approach

2.1. Trie

A *Trie*, is a special data structure used to store strings using a K-ary tree. The name *Trie* comes from the word retrieval, devised by Edward Fredkin in 1960.[5] The keys in a *Trie* are usually strings, with the characters being the edges between nodes. And the nodes in the *Trie* stores the existence of each key by marking the node. Since it does not require the sorted structure of the keys, it is possible to insert and search a key in $\mathcal{O}(N)$ time complexity, where N represents the size of the key. Here is a pseudocode for inserting data into *Trie*. 1 It takes $\mathcal{O}(N)$ time complexity, where N represents the size of the data(feature).

Algorithm 1 Insert data into a Trie

```
function INSERT(T, z)
    x ← T.root
    idx ← 0
    while idx < length(z) do
        curr ← z[idx]
        if x.children[curr] is NIL then
            x.children[curr] ← new Node()
        end if
        x ← x.children[curr]
        idx ← idx + 1
    end while
    x.data ← z
end function
```

When searching for a data in a *Trie*, we can use the following pseudocode. 2 It also takes $\mathcal{O}(N)$ time complexity.

Algorithm 2 Find data in a Trie

```
function FIND(T, z)
    x ← T.root
    idx ← 0
    while idx < length(z) do
        curr ← z[idx]
        if x.children[curr] is NIL then
            return NULL
        end if
        x ← x.children[curr]
        idx ← idx + 1
    end while
    if x.data is not NIL then
        return x.data
    else
        return NULL
    end if
end function
```

2.2. Quantization of Feature Vector, Margin

In order to utilize the *Trie* data structure, we need to quantize the feature vector into a binary vector. We quantize the feature vector by comparing each element of the vector with a threshold value of 0.5. If the value is greater than 0.5, we assign 1 to the corresponding element of the binary vector, and 0 otherwise. However, this approach has a limitation in that it does not account for variations near the threshold of 0.5. To address this limitation, we introduce a margin, denoted as ξ, allowing values in the vicinity of 0.5 : $[0.5 - \xi, 0.5 + \xi]$ to match both 0 and 1. The quantization process is illustrated in Figure 3 ($\xi = 0.1$ in this case).

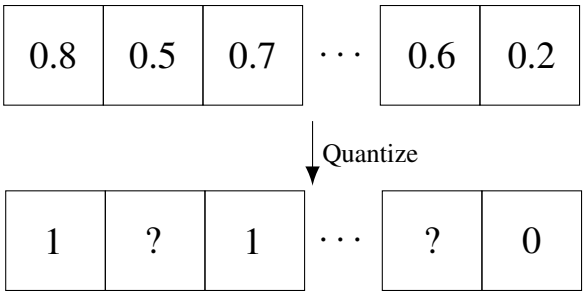

Figure 3. Quantization of vector v into ternary vector v'.

We can find quantized feature vector v' by the following pseudocode. 3

Algorithm 3 Find quantized data in a Trie

```
function FIND(T, z)
    candidates ← ∅
    q ← new queue ({T.root, 0})
    while q is not empty do
        curr, idx ← q.pop()
        if curr is NIL then
            continue
        else if idx = length(z) then
            candidates ← curr.data ∪ candidates
            continue
        else if z[idx] is ambiguous then
            q.push ({curr.children[0], idx + 1})
            q.push ({curr.children[1], idx + 1})
        else if curr.children[z[idx]] is not NIL then
            q.push ({curr.children[z[idx]], idx + 1})
        end if
    end while
    return candidates
end function
```

2.3. Features

In our approach, we conducted extensive experiments using various combinations of features to enhance the performance of our system. The selection of appropriate features plays a crucial role in accurately representing the underlying patterns and characteristics of the data. In this subsection, we describe the different features we explored and their contributions to the overall performance.

2.3.1 Mean of RGB values of N by N patches

The mean of RGB values of N by N patches feature captures the average color information within local regions of the image. By computing the mean RGB values for each patch, we can effectively represent the dominant color characteristics at different spatial scales. This feature is particularly useful for identifying objects or regions with distinct color distributions.

2.3.2 Std. Variance of RGB values of N by N patches

The standard variance of RGB values of N by N patches feature measures the degree of color variation within local regions of the image. It provides information about the diversity of color within a patch and can help distinguish between textured and non-textured regions. This feature is especially valuable for detecting areas with high color contrast or intricate patterns.

2.3.3 Corner Responses

Corner responses focus on detecting and describing corners or junctions within an image. Corner points typically indicate significant changes in image intensity or orientation, making them valuable for detecting key features, such as edges or intersections. Corner responses provide robust localization and can enhance the detection and recognition of objects with well-defined corners. Harris corner detection algorithm is well-known for corner detection.[6]

2.3.4 Blob Responses

Blob responses, on the other hand, are a type of feature that detects and characterizes regions of interest based on difference in color or brightness compared to their neighbors. Blob features are robust to certain variations such as translation, rotation since blob detection algorithm is based on image gradient.[7]

2.3.5 Haar Wavelet Responses

Haar wavelet responses are obtained by convolving the image with Haar wavelet filters. This feature representation captures local variations in both intensity and texture. Haar wavelet responses are useful for detecting edges and textures at multiple scales, allowing for robust feature extraction and discrimination. In our experiments, we evaluated the performance of our system using different combinations of these features. We conducted extensive analysis and comparisons to determine the most informative feature set that best captures the underlying characteristics of the data. The results of our experiments provide valuable insights into the effectiveness of different features and their contributions to the overall performance of our system. By considering a comprehensive set of features and their combinations, we aim to achieve a robust and accurate representation of the input data, enabling effective pattern recognition and classification in our system.[8]

3. Experiments

In this section, we will discuss of the affect of the features and the quantization margin on the performance in terms of accuracy and speed. We defined some terms to explain the results of the experiments. The terms are as follows.

- **Quantization margin** ξ: Ambigious feature value between $[0.5 - \xi, 0.5 + \xi]$ is quantized to '?'(unknown). It can be matched to 0 or 1 at *Trie*-search algorithm. We set the maximum margin as 0.3, which means that 60% of the feature value being meaningless.

- **Hash Hit - Single**: The number of cases which the data found in *Trie* by input hash is uniquely determined as the result.

- **Hash Hit - Multiple**: The number of cases which the data found in *Trie* by input hash is not uniquely determined. When multiple candidates are involved in the hash hit, the best one can be chosen by applying a naïve nearest neighbor search.

- **Accuracy**: Regardless of hash is found or not, the number of cases which the result is correct.

- **Elapsed Time**: The time taken to find all the test data.

3.1. Experiment environment

The experiment was conducted in the following environment: we utilized the **OpenCV** library for image processing using the C++ programming language. 1 To construct the database, we employed 15,000 distinct images, without considering the size of the images. Subsequently, we selected 640 images as the test data. Each test data sample was a copy of an image from the database, introducing inevitable variations from the original image due to the process of capturing photographs.

Type	Name
CPU	Intel i5-12600 @ 4.4GHz
RAM	DDR4 128GB @ 3200MHz
OS	Ubuntu 20.04 (WSL2 on Windows 10)
Language	C++17
Compiler	g++ 9.4.0
Optimization	-O3

Table 1. Experiment environment.

3.2. Feature with Mean

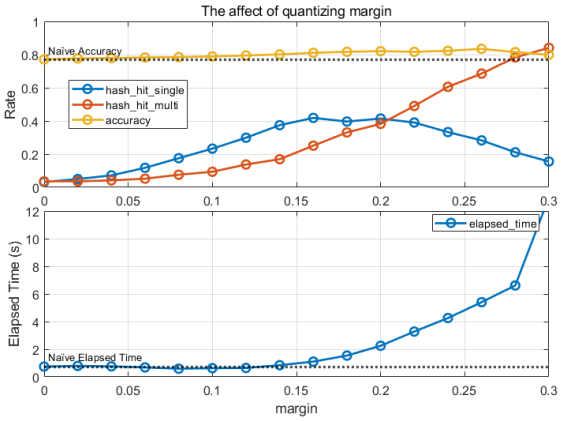

Figure 4. Feature using RGB mean of $N \times N$ patch.

Mean feature, the simplest feature among the features we cover, ought to be the most efficient and inaccurate feature. As we expected, the accuracy using the mean feature exceed the naïve nearest neighbor search slightly. Since the mean feature is not enough representive feature, the distribution of data in *Trie* is not enough balanced, which leads to the inefficiency of the *Trie* search. Elapsed time was also not much different from the naïve nearest neighbor search.

3.3. Feature with Mean and Std. Variance

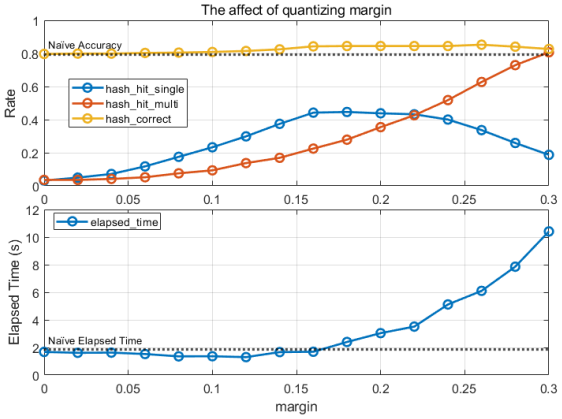

Figure 5. Feature using RGB mean and standard deviation variance of $N \times N$ patch.

To make the feature more representative, we added the standard deviation variance of RGB values of $N \times N$ patch to the mean feature so that the feature can represent the color distribution of the image. Simply the feature vector

got longer, the probability of two images having the same quantized feature vector decreased. The Single hash hit rate increased by 5%, and the multiple hash hit rate decreased by 5%. The accuracy was similar to the mean feature, but the elapsed time was slightly shorter because of the more balanced distribution of data in *Trie*.

3.4. Feature with Corner Response

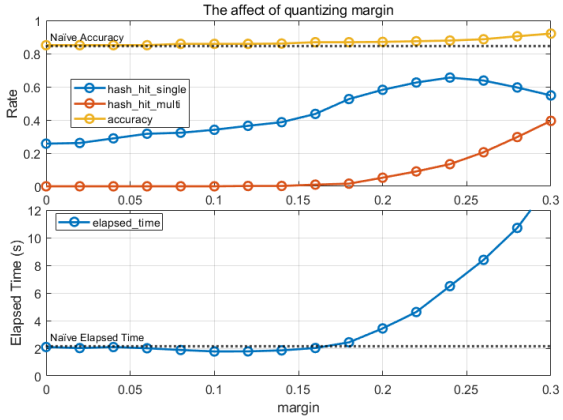

Figure 6. Feature using corner response by Harris corner detector.

To extract the corner response, we first converted the image to grayscale and then applied the Harris corner detector algorithm in **OpenCV**. The accuracy of the corner response feature was similar to naïve nearest neighbor search. The corner response makes more distinctive feature, which leads to impressively low multiple hash hit rate.

3.5. Feature with Blob Response

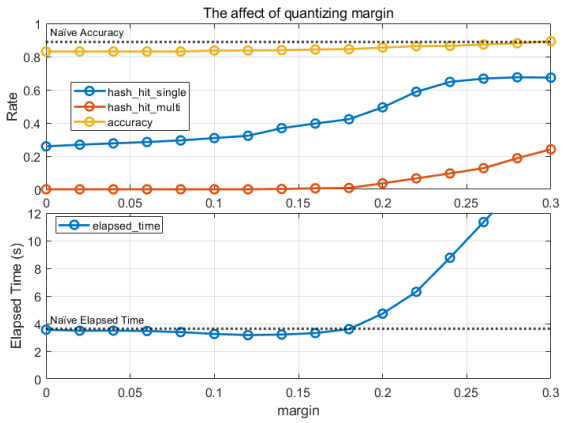

Figure 7. Feature using blob response.

To extract the blob response, we initially converted the image to grayscale and then applied the blob detector algorithm in the **OpenCV** library. We anticipated that the accuracy of the corner and blob response, considered a high-level feature, would surpass that of low-level features such as mean and standard variance.

Contrary to our expectations, the results did not align with this hypothesis. Surprisingly, the accuracy of the blob response feature was even lower than that of the naïve nearest neighbor search. In the feature extraction stage, the corner and blob response were flattened and serialized into a vector. This process may have introduced some order into the feature response. However, due to variations in lighting conditions, the response in grayscale images was not consistent, leading to an incoherent sorting order of the feature response compared to the original data.

3.6. Feature with Haar Wave Response

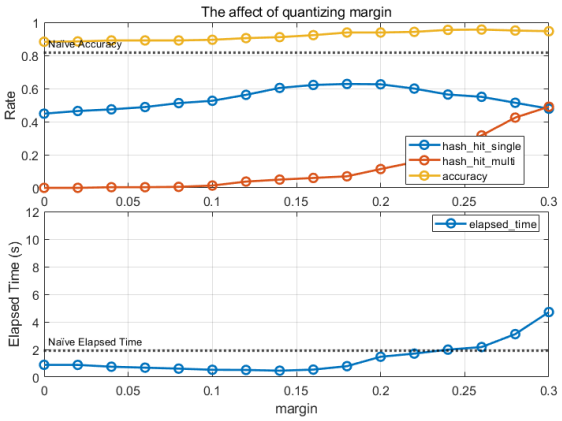

Figure 8. Feature using haar wave response.

Haar wavelet response is a kind of randomly generated feature, which is not affected by the order of the feature response. It not only captures the local feature, but also the global feature, which makes it more representative than the other features. Moreover, the haar wavelet response can be calculated rapidly by using integral image, which makes it more efficient than the other features when constructing the database. As we expected, the accuracy of the Haar wavelet response feature was the highest among the features we tested. The single hash hit case - the best case - was about 60% of the total cases, and the multiple hash hit case was about 10%. The elapsed time was also the shortest, which was about $\frac{1}{2}$ of the naïve nearest neighbor search.

3.7. Feature with N-ary *Trie* and Haar Wave Response

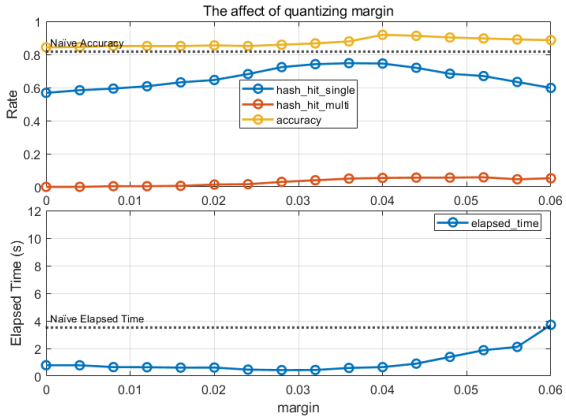

Figure 9. Feature using haar wave response and N-ary Trie. (N=5)

To enhance the balance of the *Trie* data structure, we opted to use an N-ary *Trie* instead of a binary *Trie*. Similarly, we quantized the feature vector into an N-ary vector, with a smaller quantizing margin compared to the binary case. In Figure 9, we employed a 5-ary *Trie* with a maximum quantizing margin of 0.06. The accuracy achieved was slightly higher than that of the naïve nearest neighbor search.

The single hash hit case accounted for approximately 74% of the total cases, representing the highest proportion, while the multiple hash hit case accounted for only about 4%. Increasing the number of children for each node in the tree leads to a more balanced tree structure, which in turn improves the seek time performance. The elapsed time was about $\frac{1}{4}$ of the naïve nearest neighbor search.

4. Conclusion

In this paper, we conducted several experiments using the *Trie* data structure to compare "pictures of pictures" with their original counterparts. We implemented quantization with a margin to enhance performance and applied it to various features, including mean color, variance, cornerness, blob, and Haar wavelet response.

The algorithms utilizing *Trie* demonstrated higher accuracy and faster execution time compared to the naïve nearest neighbor search. However, among the features, the blob response yielded lower accuracy when used with *Trie*, while the Haar wavelet response exhibited remarkable performance with notable execution time.

References

[1] Hyeonwoo Noh, Andre Araujo, Jack Sim, Tobias Weyand, and Bohyung Han. Large-scale image retrieval with attentive deep local features, 2018.

[2] Yoonho Hwang, Bohyung Han, and Hee-Kap Ahn. A fast nearest neighbor search algorithm by nonlinear embedding. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3053–3060, 2012.

[3] Shiv Ram Dubey, Satish Kumar Singh, and Wei-Ta Chu. Vision transformer hashing for image retrieval, 2022.

[4] Miguel Á. Carreira-Perpiñán and Ramin Raziperchikolaei. Hashing with binary autoencoders, 2015.

[5] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, sep 1960.

[6] Christopher G. Harris and M. J. Stephens. A combined corner and edge detector. In *Alvey Vision Conference*, 1988.

[7] Tony Lindeberg. Detecting salient blob-like image structures and their scales with a scale-space primal sketch: A method for focus-of-attention. *Int. J. Comput. Vision*, 11(3):283–318, dec 1993.

[8] Luc Florack, Bart ter Haar Romeny, Jan Koenderink, and M. Viergever. General intensity transformations and differential invariants. *Journal of Mathematical Imaging and Vision*, 4:171–187, 05 1994.