ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Review Article

Open Access

Enhanced image similarity search: Boundary of similarity approach with similarity hashing for Image Datasets

Sambu Patach Arrojula

USA

ABSTRACT

Image similarity searching is a well-explored topic with numerous techniques and approaches, each offering unique advantages. In this paper, we present an approach aimed at accelerating the search for the most similar images to a given query image within an image database. Our method leverages the principles of Locality-Sensitive Hashing (LSH) but relies on a concept of similarity boundaries to reach the search to most appropriate section and in turn facilitates a rapid retrieval of most similar content.We assume the availability of a reliable hash function for image similarity that offers high accuracy and a low collision rate and focus exclusively on the algorithm designed to maintain and manage similarity boundaries for each image within the hashtable. This strategic organization ensures that similar images are grouped together logically, significantly enhancing the efficiency and speed of the search process.

***Corresponding author**

Sambu Patach Arrojula, USA

Received: May 03, 2023; **Accepted:** May 10, 2023, **Published:** May 17, 2023

Keywords: Locality-Sensitive Hashing, Hash Table, Image similarity, Hamming Distance

Introduction

Popular Locality sensitive hashing has been successful and proven to help with many applications. Its main idea is to map close points in high dimensional space nearer to each other in a lower dimensional space with high probability and reduce the search space to a great extent. We here in this paper follow an approach of boundary of similarity which will be mapped/processed for the dataset that not only brings search to a very smaller subset but also has the most nearest entity readily available. Although this approach is space costly, it gives very fast and accurate results when it comes to querying similar images.

Use Case

While there are many use cases and solutions for the topic of Image similarity search following are the cases that we are considering here in this paper.

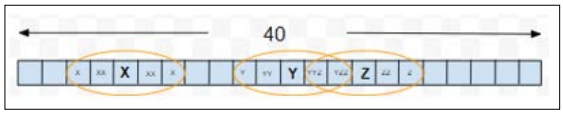
- image similarity hashes like perceptual hashing for image hash generation
- hamming distance similarity for similarity comparison

Approach

The crux of this approach is most of the applications don't have an indefinite range of similarity requirements rather have a certain degree of similarity as requirement. for example to find 95% similar images of a given image i.e. it is only interested in 95 to 100 percent similar images to a given Image not the rest. Then we define that range as a boundary of similarity and mark these per image boundaries.

Simple Linear Similarity boundary

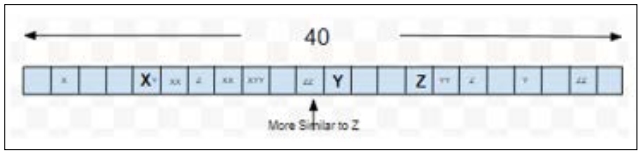
For better understanding, if the hash function generates linearly similar hashes i.e. adjacent hashes are associated with most similar keys, then the boundary marking would be illustrated as below. if X,Y and Z are the candidate entries in a 40 sized hashtable and the similarity boundary is defined as 5% then each candidate's



adjacent two indexes will be declared as similar indexes. and the similarity would fade out as the index moves away from the candidate index. and if an index is under another candidate's similarity boundary then that index will hold similarity for both candidates in such a way that it provides quick info which candidate its most similar to.

Non-Linear similarity boundaries

But in practice most of the hashes are not linear in terms of similarity. In this case we derive all the individual indexes which falls under the similarity boundary and process them i.e. update those indexes about association with candidate index/key



like shown above, the similar indexes would be scattered per key and the pointed index is more similar to Z key though it is linearly nearer to Y key. The big advantage here is that this information is readily available with all needed indexes.

Details

We consider Hamming distance when calculating similarity distance between keys.Then we design a hashtable, given 'N' as number of bits in hash(ex perceptual hash) and 'd' the desired size of similarity boundary. As preprocessing of the data set this hashtable would be used to put each image in the dataset with their corresponding generated hash.

initialization of hash table

We create a 2^N array and initialize with null. as this size will be exponentially increasing with N, there will be concerns about the size of such a huge array with bigger sizes of hash cases. We will analyse those cases further.

```
Method __init__(N, d):
  this.N = N
  this.d = d
  this.table = array of size 2^N initialized to null

Method hammingDistance(a, b):
  x = a XOR b
  dist = 0
  While x > 0:
    dist += x AND 1
    x >>= 1
  Return dist
```

Generating variations of keys under similarity boundary

The following function is designed to compute all possible variants of a given key which have a maximum specified Hamming distance. It starts by initializing an empty list for the variants and an array of bit positions from 0 to N-1, where N is the key length. The function iterates over distances from 1 to the maximum distance (maxDistance). For each distance, it uses the getCombinations function to generate all bit position combinations of that length. getCombinations works by initializing an empty list and recursively building combinations of the specified length, adding each completed combination to the result list. For each combination, generateHammingVariants creates a variant by flipping the specified bits in the original key using the XOR operation. Each variant, paired with its distance, is added to the list of variants. The function returns this list, enabling efficient management and retrieval of similar keys in the hash table.

```
Method generateHammingVariants(key, maxDistance):
    variants = empty list
    bitPositions = array of integers from 0 to N-1

    For dist from 1 to maxDistance:
      combinations = getCombinations(bitPositions, dist)
      For each combination in combinations:
        variant = key
        For each bit in combination:
          variant = variant XOR (1 << bit)
        variants.append([variant, dist])
    Return variants

  Method getCombinations(arr, k):
    result = empty list
    Helper function helper(start, comb):
      If length of comb equals k:
        result.append(copy of comb)
        Return
      For i from start to length of arr:
        comb.append(arr[i])
        helper(i + 1, comb)
        comb.pop()
    helper(0, empty list)
    Return result
```

Inserting a key-value

The put method inserts a key-value pair into the hash table, ensuring that both the exact key and its Hamming variants are updated. when updating a slot it creates a pair of value and hamming-distance of the slot from the given key original slot. If the exact key slot in the table is empty, it initializes it with the value-distance pair. If not, it adds the pair to the existing list, sorting it by distance to prioritize closer matches. The method then generates Hamming variants of the key up to the specified maximum distance using the generateHammingVariants function. This process ensures that any slot will be having information about what all the values (Images) it is nearer/similar that too sorted by similarity in turn helps efficient and quick retrieval of most similar Images for a given hash (say query Image).

```
Method put(key, value):
    If key is invalid:
      Print error message
      Return

    If this.table[key] is null:
      this.table[key] = [{value: value, distance: 0}]
    Else:
      this.table[key].append({value: value, distance: 0})
      Sort this.table[key] by distance

    variants = generateHammingVariants(key, this.d)
    For each [variant, dist] in variants:
      If this.table[variant] is null:
        this.table[variant] = [{value: value, distance: dist}]
      Else:
        this.table[variant].append({value: value, distance: dist})
        Sort this.table[variant] by distance
```

Deletion of a Certain Key

The delete method removes a key-value pair from the hash table and its Hamming variants. If the key slot is empty, the method returns immediately. Otherwise, it searches for the entry with the given value in the key slot. If found, and its distance is zero (indicating an exact match), the method generates Hamming variants of the key using generateHammingVariants. It then iterates through these variants, removing the value entry from each variant slot. If a variant slot becomes empty after removal, it is set to null. The method also removes the value entry from the exact key slot and sets it to null if it becomes empty. This ensures that the key-value pair and all its similar variants are thoroughly deleted from the hash table.

```
Method delete(key, value):
    If key is invalid:
      Print error message
      Return

    If this.table[key] is null:
      Return

    Find the entry with the given value in this.table[key]
    If entry with distance 0 is found:
      Print found exact match message
      variants = generateHammingVariants(key, this.d)
      For each [variant] in variants:
        If this.table[variant] is not null:
          Remove entry with given value from this.table[variant]
          If this.table[variant] is empty:
            Set this.table[variant] to null

    Remove entry with given value from this.table[key]
    If this.table[key] is empty:
      Set this.table[key] to null
```

Query An Exact or Nearest Value to A Given Key

The get method retrieves the value associated with a given key from the hash table. If the key slot in the table is empty, the method returns null, indicating that the key is not present in the hash table. If the key slot is not empty, the method returns the value associated with the key, specifically the first entry in the list for that key. This ensures that the most relevant value, typically the one with the shortest distance, is retrieved efficiently. The get method provides a straightforward way to access the stored values based on their

exact keys.

```
Method get(key):
    If key is invalid:
      Print error message
      Return null

    If this.table[key] is null:
      Return null
    Else:
      Return this.table[key][0].value
```

Analysing time complexity of operations of this hash-table
Get operation
Under ideal conditions, get operations are extremely fast and can be completed in constant time. Assuming the entire hashtable is stored in memory, retrieving an entry for a given key would have a time complexity of O(1). Additionally, obtaining the least distance value for that entry from an ordered list should also be O(1). However, other operations such as insertion (put) and deletion are significantly more time-consuming. Let's analyze these operations further.

Put operation:
- **Exact Key Insertion:** Inserting the exact key into the hash table involves checking if the slot is null and adding the key-value pair. This operation is O (1).
- **Generating Hamming Variants:** The primary cost in the put method comes from generating Hamming variants. The number of variants is determined by the combinations of bit positions for distances from 1 to maxDistance (d).

o Generating all combinations of k bit positions from N has a complexity of $O(\binom{N}{d})$.
o The total number of variants across all distances up to d is the sum of combinations for each distance, resulting in $O(\sum_{k=1}^{d}\binom{N}{k})$
o For small d relative to N, this can be approximated as $O(N^d)$.

- **Inserting Variants:** Each generated variant is inserted into the hash table, which is O(1) per insertion.
- **Sorting:** After inserting the variants, sorting is performed, which, if using a typical sorting algorithm like QuickSort or MergeSort, has a complexity of O(m log m), where m is the number of entries in the slot. However, since we only insert a fixed number of items (d variants), this can be considered O(1) in the context of each put operation.

Total Time Complexity for put: $O(N^d)$.
Deletion (delete):
- **Exact Key Deletion:** Similar to insertion, checking and deleting the exact key from the hash table is O(1).
- **Generating Hamming Variants:** The cost for generating Hamming variants is the same as in the put method: $O(N^d)$.
- **Deleting Variants:** Each generated variant is checked and the specific value is deleted. This is O(1) per deletion, but since there are up to N^d variants, this results in $O(N^d)$ deletions.

Total Time Complexity for delete: $O(N^d)$.
As we can see these are concluding to exponential operations, but obviously making the get() function very efficient.

Analyzing space complexity of this hash table
Storage per Key:
- For the exact key, we store the key-value pair directly, which is O(1) space.
- Each key can generate up to $O(N^d)$ variants, and each variant is stored in a separate slot with its distance information.
- The space needed for storing these variants is therefore .

Storage for the Hash Table: $O(N^d)$.
- The hash table itself has 2^N slots.
- Each slot can hold multiple key-value pairs, especially considering variants.

Space Complexity for the Entire Table: $O(2^N.N^d)$
This is an extremely high space constraint, a simple image hash with 32 bit and 5 bit hamming distance similarity boundary could end up around 15,200 terabytes.

Next Steps
To address the space and time complexities, several strategies can be considered:
1. Disk-backed Hash Tables: Implementing disk-backed hash tables can help manage space requirements. This approach would involve loading only the necessary parts of the table into memory, with the rest being stored on disk. This allows for handling larger datasets by leveraging external storage, thus reducing the in-memory space requirements. this will hit performance for hash table operations but would make it feasible for other data sets and bigger hash cases
2. **2Parallel Processing:**

o **For put():** Implement slot-level locking for exclusive access before writing or updating the entry (list of values) in the actual slot and all slots within the similarity boundary. This ensures that concurrent insertions do not interfere with each other, maintaining data integrity.
o **For delete():** Implement key-level locking for exclusive access when updating all slots under its similarity boundary. This would require changing the entry structure from {value, distance} to {value, key, distance} to ensure that the value is accessed only if the key is not locked.

These enhancements aim to improve the efficiency and scalability of the hash table, making it more suitable for larger datasets and concurrent operations. By leveraging disk storage and parallel processing, the system can handle larger amounts of data while maintaining the integrity and performance of the hash table operations.

Conclusions
The HammingSimilarity-HashTable implementation showcases an impressive and efficient time complexity for the get() function, operating in constant time O (1). This ensures rapid retrieval of values, making it ideal for scenarios where quick access to data is crucial. However, the time complexity of the put() and delete() functions is significantly higher, approximately $O(N^d)$, due to the need to generate and manage numerous Hamming variants. While these operations are computationally intensive, they are typically less frequent than retrieval operations, making the approach suitable for fixed datasets where preprocessing of insertions can be done in advance, leaving only the get () function to be used in real-time.

Nonetheless, the space complexity of the hash table is a major concern, growing exponentially with the size of the hash (N) and the similarity boundary (d). This exponential space requirement makes the current implementation feasible only for small-scale, fixed image datasets, where the number of possible keys and their variants can be reasonably managed [1-7].

References
1. https://pyimagesearch.com/2014/02/17/building-an-image-search-engine-defining-your-similarity-metric-step-3-of-4/
2. Different types of distance metrics in machine learning
3. Locality Sensitive Hashing (LSH): The Illustrated Guide
4. Random Projection for Locality Sensitive Hashing

5. Perceptual Hashes: measuring similarity

6. Similarity hashing and perceptual hashes