Combining Perceptual Hashing Based on Hamming DIstance and BK Tree Indexing for Efficient Image Similarity Detection

Steven Tan - 13524060
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: steventan6002@gmail.com , 13524060@std.stei.itb.ac.id

***Abstract*—This paper addresses the inefficieny of linear search compared to more efficient searching allowed by metric data tree and perceptual hashing.**

***Keywords*—perceptual hash, bk tree**

I. Pendahuluan

Recent technology advancement has led to an unprecedented explosion in the volume of visual data. Billions of images are uploaded daily to social media, e-commerce sites, and cloud storage services. This has created a pressing need for systems that can efficiently search these images. Content-Based Image Retrieval (CBIR) has emerged as a key technology to address this challenge. CBIR helps users find images based on their content rather than relying on words decription or metadata.

A fundamental task in CBIR is similarity detection. Similarity detection is about finding images that are visually similar or near-duplicates of a queried image. A naive approach involves a pixel by pixel comparison, which is not only slow but also susceptible. Minor modifications such as resizing, compression, or slight color adjustments can misidentify two visually identical images to be completely different when examined pixel by pixel.

Perceptual hashing algorithm have been developed to address these limitations. Unlike cryptographic hashes which are sensitive to even minor bit changes, perceptual hashes are robust to minor, perceptually insignificant modifications. Similar images will have similar hashes. The similarity between these hashes can be measured using the Hamming distance, the number of differing bits in their binary representations.

However, even with fast hash comparisons, searching a database of millions or billions of images by calculating the Hamming distance against every hashes could result in significant time spent. This paper intends to analyse an alternative solution that addresses this scalability issue by indexing the perceptual hashes in a BK-Tree, a metric tree data structure designed for efficient fuzzy searching. This combination allows for a system that is both robust in detecting similarity and efficient in its searching time.

II. Theoretical Framework

A. Perceptual Hashing

Perceptual hashing is a type of hashing algorithm that intends to maximize hash collision. This results in same or similar hashes being produced for inputs that are similar. This means we only need to check two inputs hashes to see if they are similar or not. This also allows it to be robust against minor modification made to the input. But, its strength is not only limited to that

Image, when extracted into data, tends to be stored in a 2D matrix format with pixel as its element. Although 2D matrix is an intuitive data structure for this, doing a comparison for two image is tedious and time consuming. So other ways of storing image data was developed. One of those is storing it in a !D matrix or an ordered list of pixel.

A 1D matrix representation allows images to be treated as a single vector or point with every single element, pixel, as the dimensions. While vectors or points are easier to deal with, it also introduces problem. One of these is called the curse of dimensionality. With each pixel being treated as a dimensions, this means an image of N x M size will have N x M dimensions.

The curse of dimensionality is bad because higher dimensions results in distance between any two points becoming more similar to each other. This means the farthest distance between any two points could be almost as close to the nearest distance between any two points. This means if images representated as data points were to be analyzed its similarity, using Hamming distance to compare them would not be possible as the farthest and nearest distance could just differ by one.

This is where perceptual hashing comes in. Since perceptual hashing always reduce any images to just X bit hashes. This effectively reduces an image dimensions of N x M to just X dimensions. This also allows it to be compared with metric distances like Hamming distance.

There are several algorithms that is included in this family, such as difference hashing, average hashing, pHash, wavelet hash etc. Difference hashing will be the algorithm used and focused on in this paper.

In this paper, difference hash is implemented by converting the image to a greyscale and reducing the size of the image to 9 x 8. Each pixel in a row will be compared with the order of left to right. This will continue until the bottom row is reached. Two adjacent pixel will determine the hash bits. If the left pixel is smaller in value than the right pixel then one will be appended to the end of the hash bits. Else zero will be appended.[1]

B. Hamming Distance

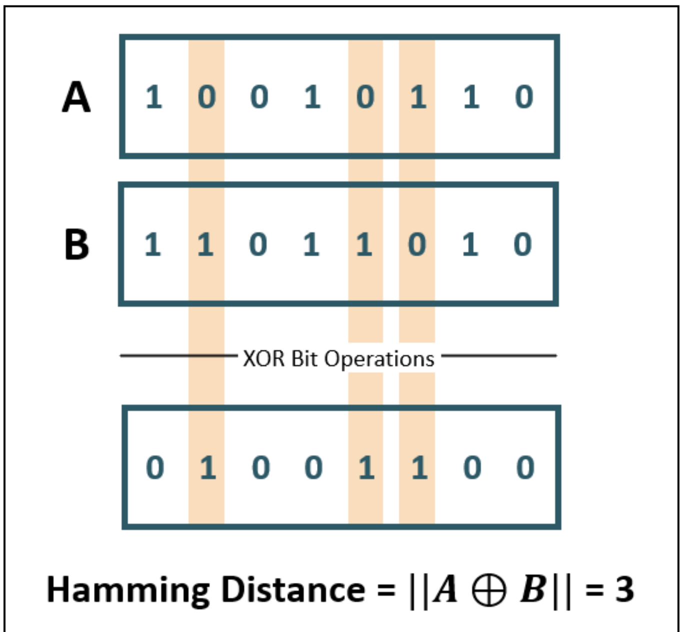


Fig. 1. Hamming Distance Example Source : https://docs.oracle.com/en/database/oracle/oracle-database/23/vecse/hamming-distance.html

Hamming distance is a metric used to measure the edit distance between two strings or vector of equal length. Hamming distance measures only the substitution needed to make two strings or vector equal. Hamming distance holds an important property that is it satisfies the triangle inequality. Its importance is related to BK Tree, a type of metric tree.

$$d(x,y) \le d(x,z) + d(z,y) \tag{1}$$

C. Metric Tree

A metric tree is a tree for indexing data in metric spaces. This type of tree takes advantage of metric spaces' properties such as triangle inequality to do efficient data accessing. A Burkhard-Keller Tree (BK Tree) is a type of metric tree.

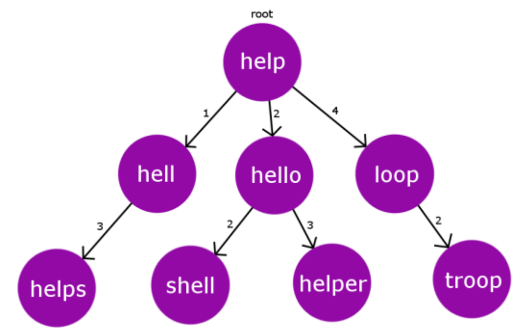


Fig. 2. BK Tree example Source : BK-Tree | Introduction & Implementation - GeeksforGeeks

Any string or vector can be chosen as the root. The child of a parent node is determined by first checking if there is already a child node with the same distance.

$$d(\mathrm{a},\mathrm{p}) = d(\mathrm{p},\ \mathrm{c}) \tag{2}$$

With d(x, y) being the distance function that returns the distance between two nodes, a being the node to be inserted, p being the parent node and c being the child node.

If there is already a child node with the same distance, then we traverse to a subtree with that child node being the new root. This process repeats until we find that the distance between the node to be inserted and the parent node isn't the same with the distance between any children node and the parent node. That node is then inserted as a new node.

The triangle of inequality shows its uses in searching a BK Tree. Let q to represent the queried node, p to represent the parent node and c to represent the child node. Then, from equation(1) we derive the following equation

$$d(\mathrm{p},\mathrm{q}) \le d(\mathrm{p},\mathrm{c}) + d(\mathrm{c},\mathrm{q}) \tag{3}$$

$$d(\mathrm{p},\mathrm{q}) - \mathrm{d}(\mathrm{p},\mathrm{c}) \le d(\mathrm{q},\mathrm{c}) \tag{4}$$

The next two equations can also be derived from equation(1), just the pairing orders are switched.

$$d(\mathrm{p},\mathrm{c}) \le d(\mathrm{p},\mathrm{q}) + d(\mathrm{q},\mathrm{c}) \tag{5}$$

$$d(\mathrm{p},\mathrm{c}) - \mathrm{d}(\mathrm{p},\mathrm{q}) \le d(\mathrm{q},\mathrm{c}) \tag{6}$$

Notice that equation(4) and (6) is similar in the left hand side.

$$d(\mathrm{p},\mathrm{q}) - \mathrm{d}(\mathrm{p},\mathrm{c}) = -(d(\mathrm{p},\mathrm{c}) - \mathrm{d}(\mathrm{p},\mathrm{q})) \tag{7}$$

From equation(7) we can conclude the following equation

$$d(\mathrm{q},\mathrm{c}) \geq |d(\mathrm{p},\mathrm{q}) - d(\mathrm{p},\mathrm{c})| \qquad (8)$$

If we set a cutoff or tolerance n, then we mean that the distance between queried node and the child must be less than n.

$$d(\mathrm{q},\mathrm{c}) \leq \mathrm{n} \qquad (9)$$

If the above equation is true, then we can say that the two nodes, q and c, are similar. And with equation (8) and (9) we can derive the following equation.

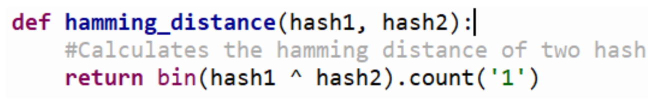
$$d(\mathrm{p},\mathrm{q}) - \mathrm{n} \leq d(\mathrm{p},\mathrm{c}) \leq d(\mathrm{p},\mathrm{q}) + \mathrm{n} \qquad (10)$$

The above equation allows us to know which child node of a parent node is similar to the queried node.

III. IMPLEMENTATION

A. Implementing Hamming distance calculator

As the example in figure 1, a clever trick to computing the Hamming distance is to XOR out the same bits that are in the same position. Then, we can count the remaining 1 that are left.

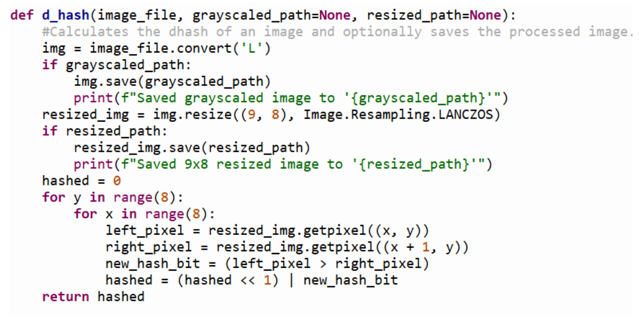


```python
def hamming_distance(hash1, hash2):
    #Calculates the hamming distance of two hash
    return bin(hash1 ^ hash2).count('1')
```

Fig. 4. Implementation of Hamming distance calculator in Python

B. Implementing difference hashing

Difference hash(dHash) will be implemented with the help of the python library pillow. Pillow, the image module, helps is extracting the pixel value of an image, converting an image to grayscale and resizing an image. This allows the code to contain only the hashing computation, thus simplifying it.

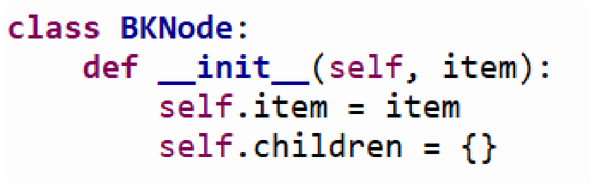


```python
def d_hash(image_file, grayscaled_path=None, resized_path=None):
    #Calculates the dhash of an image and optionally saves the processed image.
    img = image_file.convert('L')
    if grayscaled_path:
        img.save(grayscaled_path)
        print(f"Saved grayscaled image to '{grayscaled_path}'")
    resized_img = img.resize((9, 8), Image.Resampling.LANCZOS)
    if resized_path:
        resized_img.save(resized_path)
        print(f"Saved 9x8 resized image to '{resized_path}'")
    hashed = 0
    for y in range(8):
        for x in range(8):
            left_pixel = resized_img.getpixel((x, y))
            right_pixel = resized_img.getpixel((x + 1, y))
            new_hash_bit = (left_pixel > right_pixel)
            hashed = (hashed << 1) | new_hash_bit
    return hashed
```

Fig. 3. dHash implementation in Python

C. Implementing BK Tree

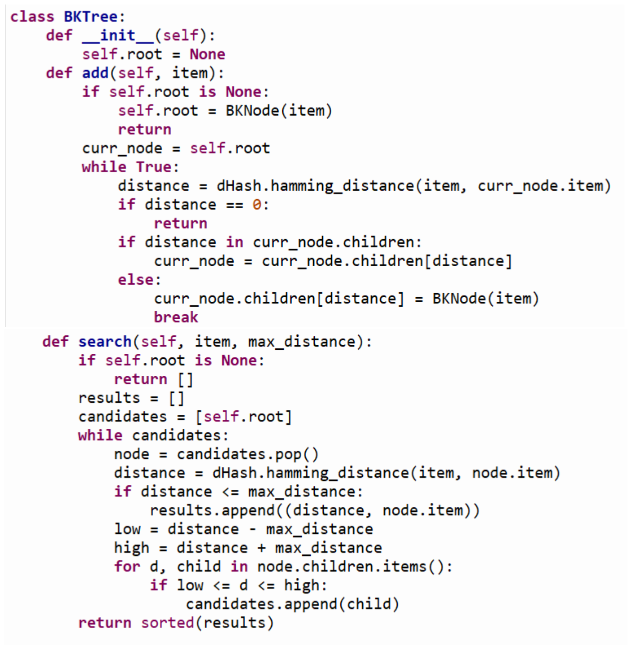
Each BK Tree Node will contain an item or value which is the dHash value and a children. The children component is implemented using dictionary since dictionary holds one property that makes it fitting for this task. That is each key must be unique from one another. Since the children of a node can't possibly have the same edge value this means the edge value can be stored as the key, while the value will be the child node itself.



```python
class BKNode:
    def __init__(self, item):
        self.item = item
        self.children = {}
```

Fig. 5. Implementation of BK Node in Python

The BK Tree implementation will be as discussed in the theoretical framework for metric tree.

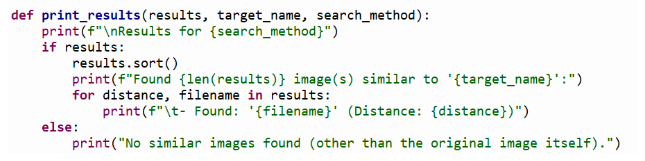


```python
class BKTree:
    def __init__(self):
        self.root = None
    def add(self, item):
        if self.root is None:
            self.root = BKNode(item)
            return
        curr_node = self.root
        while True:
            distance = dHash.hamming_distance(item, curr_node.item)
            if distance == 0:
                return
            if distance in curr_node.children:
                curr_node = curr_node.children[distance]
            else:
                curr_node.children[distance] = BKNode(item)
                break
    def search(self, item, max_distance):
        if self.root is None:
            return []
        results = []
        candidates = [self.root]
        while candidates:
            node = candidates.pop()
            distance = dHash.hamming_distance(item, node.item)
            if distance <= max_distance:
                results.append((distance, node.item))
            low = distance - max_distance
            high = distance + max_distance
            for d, child in node.children.items():
                if low <= d <= high:
                    candidates.append(child)
        return sorted(results)
```

Fig. 6. Implementation of BK Tree in Python

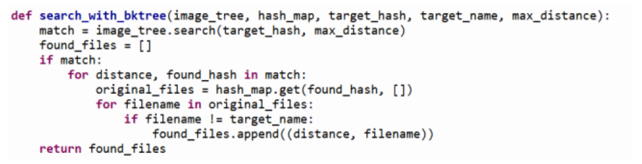
IV. TESTING AND ANALYSIS

For the testing, module image from pillow library, os module, time module and glob module will be imported to help testing. os and glob module are imported to help in loading and saving file images, while time is imported to help know the search time with BK Tree and brute force.



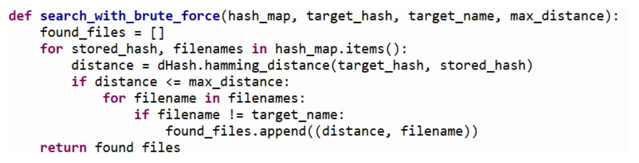
```python
def print_results(results, target_name, search_method):
    print(f"\nResults for {search_method}")
    if results:
        results.sort()
        print(f"Found {len(results)} image(s) similar to '{target_name}':")
        for distance, filename in results:
            print(f"\t- Found: '{filename}' (Distance: {distance})")
    else:
        print("No similar images found (other than the original image itself).")
```

Fig. 7. Implementation of displaying similar images filename in Python



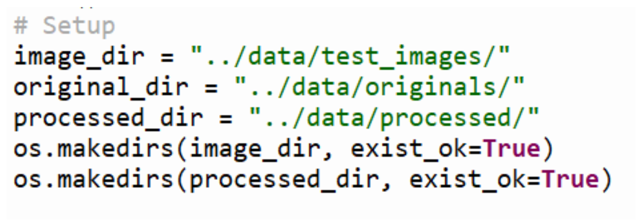
```python
def search_with_bktree(image_tree, hash_map, target_hash, target_name, max_distance):
    match = image_tree.search(target_hash, max_distance)
    found_files = []
    if match:
        for distance, found_hash in match:
            original_files = hash_map.get(found_hash, [])
            for filename in original_files:
                if filename != target_name:
                    found_files.append((distance, filename))
    return found_files
```

Fig. 9. Implementation of image similarity searching with BK Tree in Python



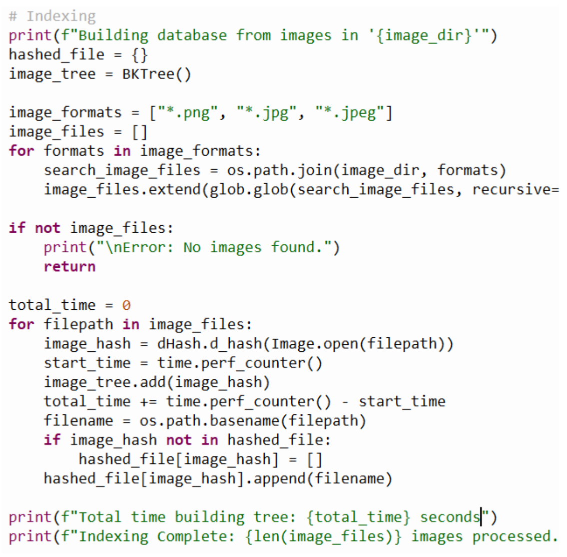
```python
def search_with_brute_force(hash_map, target_hash, target_name, max_distance):
    found_files = []
    for stored_hash, filenames in hash_map.items():
        distance = dHash.hamming_distance(target_hash, stored_hash)
        if distance <= max_distance:
            for filename in filenames:
                if filename != target_name:
                    found_files.append((distance, filename))
    return found_files
```

Fig. 8. Implementation of image similarity brute force searching in Python



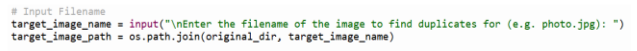
```python
# Setup
image_dir = "../data/test_images/"
original_dir = "../data/originals/"
processed_dir = "../data/processed/"
os.makedirs(image_dir, exist_ok=True)
os.makedirs(processed_dir, exist_ok=True)
```

Fig. 10. Implementation of testing setup in Python



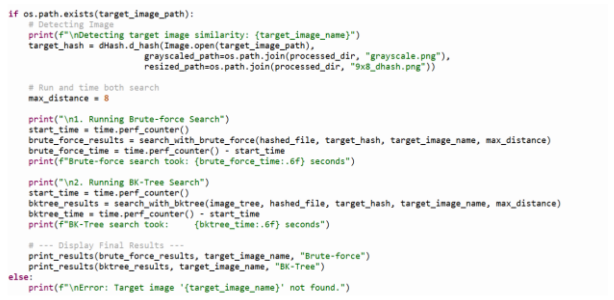
```python
# Indexing
print(f"Building database from images in '{image_dir}'")
hashed_file = {}
image_tree = BKTree()

image_formats = ["*.png", "*.jpg", "*.jpeg"]
image_files = []
for formats in image_formats:
    search_image_files = os.path.join(image_dir, formats)
    image_files.extend(glob.glob(search_image_files, recursive=

if not image_files:
    print("\nError: No images found.")
    return

total_time = 0
for filepath in image_files:
    image_hash = dHash.d_hash(Image.open(filepath))
    start_time = time.perf_counter()
    image_tree.add(image_hash)
    total_time += time.perf_counter() - start_time
    filename = os.path.basename(filepath)
    if image_hash not in hashed_file:
        hashed_file[image_hash] = []
    hashed_file[image_hash].append(filename)

print(f"Total time building tree: {total_time} seconds")
print(f"Indexing Complete: {len(image_files)} images processed.
```

Fig. 11.
Implementation of indexing image files in Python



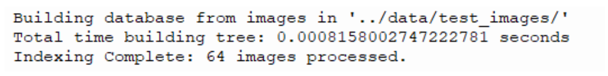
```python
# Input Filename
target_image_name = input("\nEnter the filename of the image to find duplicates for (e.g. photo.jpg): ")
target_image_path = os.path.join(original_dir, target_image_name)
```

Fig. 12. Implementation of image filename input in Python



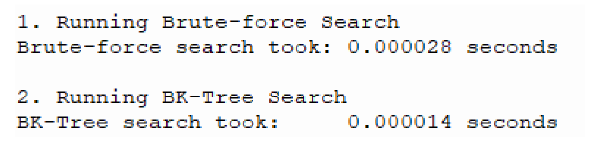
```python
if os.path.exists(target_image_path):
    # Detecting Image
    print(f"\nDetecting target image similarity: {target_image_name}")
    target_hash = dHash.d_hash(Image.open(target_image_path),
                               grayscaled_path=os.path.join(processed_dir, "grayscale.png"),
                               resized_path=os.path.join(processed_dir, "9x8_dhash.png"))

    # Run and time both search
    max_distance = 8

    print("\n1. Running Brute-force Search")
    start_time = time.perf_counter()
    brute_force_results = search_with_brute_force(hashed_file, target_hash, target_image_name, max_distance)
    brute_force_time = time.perf_counter() - start_time
    print(f"Brute-force search took: {brute_force_time:.6f} seconds")

    print("\n2. Running BK-Tree Search")
    start_time = time.perf_counter()
    bktree_results = search_with_bktree(image_tree, hashed_file, target_hash, target_image_name, max_distance)
    bktree_time = time.perf_counter() - start_time
    print(f"BK-Tree search took:     {bktree_time:.6f} seconds")

    # --- Display Final Results ---
    print_results(brute_force_results, target_image_name, "Brute-force")
    print_results(bktree_results, target_image_name, "BK-Tree")
else:
    print(f"\nError: Target image '{target_image_name}' not found.")
```

Fig. 13. Implementation of testing and comparing both search in Python



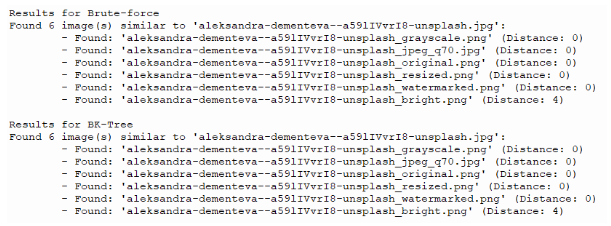
```
Building database from images in '../data/test_images/'
Total time building tree: 0.0008158002747222781 seconds
Indexing Complete: 64 images processed.
```

Fig. 14. Time took building trees and total images indexed



```
1. Running Brute-force Search
Brute-force search took: 0.000028 seconds

2. Running BK-Tree Search
BK-Tree search took:     0.000014 seconds
```

Fig. 15. Time comparison between both method of searching



```
Results for Brute-force
Found 6 image(s) similar to 'aleksandra-dementeva--a59lIVvrI8-unsplash.jpg':
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_grayscale.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_jpeg_q70.jpg' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_original.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_resized.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_watermarked.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_bright.png' (Distance: 4)

Results for BK-Tree
Found 6 image(s) similar to 'aleksandra-dementeva--a59lIVvrI8-unsplash.jpg':
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_grayscale.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_jpeg_q70.jpg' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_original.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_resized.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_watermarked.png' (Distance: 0)
        - Found: 'aleksandra-dementeva--a59lIVvrI8-unsplash_bright.png' (Distance: 4)
```

Fig. 16. Image similarity results for both search method

Before discussing the code, I would like to mention that the dataset were taken from unsplash.com and modified to test different cases.

First the time, as expected brute force search takes about twice as long BK Tree searching. While this doesn't matter much since it is shorter than even 1 millisecond. As data grew this difference will become more apparent. And not just apparent too but also larger if we take a look at their Big O notation respectively.

Brute force search has about O(n), linear scaling, compared to BK Tree O(log(n)), logarithmic scaling. This means that as more data is being indexed, searching with BK Tree becomes much more efficient.

Though it doesn't mean there isn't downsides to this. While BK Tree is faster, it comes with additional time and space cost. As you can see in figure 12, there is time taken building BK tree. It is insignificant now but when more data is being indexed, it will take longer too. Same with the extra space to build a tree.

While dHash is a good algorithm to detect image similarity, it still fails to detect cropped and rotated image. Out of 8 similar image, it only manage to detect 6, one of which is an exact duplicate.

V. Conclusion

Using BK tree to index image hashes has proven to be a solution to efficient image similarity searching. Instead of checking one by one hashes, BK tree allows searching in a pattern that proves to be fast. Triangle inequality allows BK tree to have an effective and efficient pruning logic when searching.

While dHash fails in detecting cropped and rotated image, it still manages to hash it in a way so it can detect other modifications to an image. It remains a good, simple and effective image deduplication algorithm.

There still remains the challenge of implementing a more effective and image similarity algorithm. For indexing, one could try to not rely on metric distance, in particular hamming distance, for efficient searching and see if it could be more efficient.

Appendix

Github repository for project: StevenT-1/Perceptual-Hashing-and-BK-Tree-for-Makalah

Acknowledgment

I would like to express sincere gratitude to God Almighty for the guidance and comfort in writing this paper. I would also like to give special thanks to Arrival Dwi Sentosa, S.Kom., M.T., for his role as the lecturer in the IF1220 Discrete Mathematics course. Additionally, the author would like to thank Dr. Ir. Rinaldi Munir, M.T., for publishing the lecture materials on the website. Lastly, I would like to give special thanks to unsplash.com for providing the image needed for free.

References

[1] N. Krawetz, "Kind of like that...," *HackerFactor*, May 21, 2013. [Online]. Available: https://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html. [Accessed: Jun. 17, 2025].

[2] N. Johnson, "Damn cool algorithms: part 1 - BK-Trees," *notdot.net*, Apr. 21, 2007. [Online]. Available: http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees. [Accessed: Jun. 17, 2025].

[3] R. Munir, "Pohon (Bagian 1)," *Matematika Diskrit Course Notes, Institut Teknologi Bandung*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [Accessed: Jun. 17, 2025].

[4] R. Munir, "Pohon (Bagian 2)," *Matematika Diskrit Course Notes, Institut Teknologi Bandung*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf. [Accessed: Jun. 17, 2025].

[5] R. Munir, "Teori bilangan (Bagian 3)," *Matematika Diskrit Course Notes, Institut Teknologi Bandung*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/17-Teori-Bilangan-Bagian3-2024.pdf. [Accessed: Jun. 17, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

[illegible]

Steven Tan(13524060)