Efficient Nearest Neighbors Search in Distributed Manner

Cheng, Ti-Chung
Chinese University of Hong Kong
Department of Computer Science and Engineering
tcheng@link.cuhk.edu.hk

Supervised by Prof. James Cheng
Chinese University of Hong Kong
Department of Computer Science and Engineering
jcheng@cse.cuhk.edu.hk

Abstract

Nearest neighbor Search has been one of the most interesting optimization problems when it comes to large data sets. Image retrieval is yet one of the application of nearest neighbor search problems. In this project, we present our understanding of the problem and our step-by-step methods to tackle them. Our goal is to ultimately demonstrate the application of a distributed nearest neighbor search application. We would visit feature selection, feature matching and nearest neighbor search on single and multiple machines and finally introduce the distributed nearest neighbor search model.

1 Introduction

Image retrieval is one of the most used application areas when discussing the problem of Nearest Neighbor Search (NNS). This is a question that states when given a segment or a transformed segment of a particular image, find the most correlated image, or the original image from stock images. Given that images are stored with high dimensionality data, it is one of the most challenging and interesting topics in data mining and machine learning.

From the above definition, we can see that nearest neighbor search is the special case of K-nearest neighbor search (K-NNS) when k equals one. K-NNS focuses on returning the first k nearest results instead of just the nearest one. K-NNS has been a long yet on-going research problem that is involved in numerous discussions [1, 2]. Various methods such as kd-tree [3], R-tree [2], ball cover [1] are different approaches for tackling this problem. However, datasets, such as images, have a high dimensionality, which leads to the problem "curse of dimensionality"[4]. The"curse of dimensionality" problem describes that the computation resources required for nearest neighbor search matching grows exponentially when given high dimensional data.

To tackle this problem, one of the most popular methods is an unsupervised learning algorithm known as Locality Sensitive Hashing, or LSH[4]. It is a hashed based algorithm that requires no labeling in the first place. Though LSH aims at tackling the ε-approximate nearest neighbor search problem (ε-ANN), without loss of generality, the algorithm performs well for high dimensional data nearest neighbor

search. Yet, as the stock images, or data in general, grow, it becomes difficult for a single machine to handle and it faces challenges in efficient computation. Thus, it is important for us to perform and design frameworks for distributed computing that enables user to conduct efficient and even effective searches within different disciplines by implementing a general use framework for nearest neighbor search.

In this final year project report, we should follow this logic to explain, revisit and demonstrate our project conduct and evaluations. In section 2, we will first provide a brief review of background and related works. Section 3 to 7 will cover the detailed implementation of the entire project. In section 3, we will introduce our framework of a similar image retrieval system using nearest neighbor search on a single machine. Section 4 demonstrates the implementation of E2LSH on a single machine for image retrieval. By increasing our amount of data in data sets, we will introduce how we transform E2LSH on a single machine into a distributed environment in section 5. Section 6 and 7 will include our generalization of such distributed framework and the implementation of several other LSH nearest neighbor search algorithms. Cross evaluation and discussion will be provided in section 8, followed by our conclusion and future work in section 9.

2 Background and Related Works

2.1 Nearest Neighbor Image Search

Image retrieval is one of the most significant application of nearest neighbor search problems. There has been various strategies in image retrieving including content-based image retrieval [5] as well as concept-based approaches [6] or a mixture of both. The former aims at feature extraction by getting the information from images through data representation such as colors, shapes and any other information that is directly involved within the image itself. The latter is the concept of meta-data extraction such as keywords and object recognition extracted from a given image. In our current project, we focused on the first type of extraction method, or specifically the Scale Invariant Feature Transform (SIFT) [7] which we would show demonstrations of such transformation in the next section. SIFT descriptors allow the preservation of the data for points of interests of any given image despite illumination and geometric transformation such as scaling and rotation by encoding local gradient patterns [7].

By incorporating such high dimensional data descriptor, we would require $\varepsilon-$approximate nearest neighbor search ($\varepsilon-$ANNS or ANNS) instead of exact nearest neighbor search to reduce excessive computational and memory costs. Both definitions should be formally described as follows:

Definition 1: Nearest neighbor Search

Retrieving a set of $k-$most similar items as set $R \subset X$ when given a query q, where the superset X includes n points, $X = \{x_i \mid i = 1...n\}$ and $x_i \in R^d$

Definition 2: $\varepsilon-$approximate nearest neighbor search

An approximated version of nearest neighbor search where given a upper bound of $\varepsilon > 0$ as the

error ratio given the entire data set X, such that point $p \in X$ is an $\varepsilon-$approximate nearest neighbor of query point q if and only if distance $d(p,q) \leq (1+\varepsilon)d(p*,q)$ where $p*$ is the retrieved nearest neighbor.

2.2 Locality Sensitive Hashing

When it comes to ANNS algorithms, Locality Sensitive Hashing (LSH)[8] is arguably the state-of-the-art method with promising results. It is a hashed based clustering method for unsupervised learning with data-independent dependencies[9]. However, the original design of LSH is to solve the (*R*,*c*)-nearest neighbor problem. This problem specifies that given a query data point as the center of a given ball, how should we select a function h from the function family H such that it fulfills the following definition:

Definition 1 *Ball $B_p(\overrightarrow{q},r)$: In d-dimensional space R^d, under distance metric l_p, B_p is centered at point $\overrightarrow{q}$ with radius r such that : $BallB_p(\overrightarrow{q},r) = \{\overrightarrow{v} \in R^d \mid l_p(\overrightarrow{r},\overrightarrow{q}) \leq r\}$*

When the following conditions were met we would then define this family function H as *(r, cr, p1, p2)*-sensitive:

Definition 2 (Locality-sensitive Hashing)

1. *if $d(q,r) \leq r$, then $P_r[h(p) = h(q)] \geq p_1$*
2. *if $d(q,r) > cr$, then $P_r[h(p) = h(q)] \leq p_2$*
3. $c \geq 1$ *and*
4. $p_1 > p_2$

Though LSH does not directly return the top k results of a given nearest neighbor search, it tries to return the data points that are covered within the given ball by a given radius c with an error threshold ε . LSH hash functions enable the power to search through a growing radius r without the loss of generality. Through the alternation of this function under the same function family H, we are able to capture the data points with high probability that are similar to the given query data. With this in mind, we then define different distance metrics from different metric spaces and optimization to perform different searches due to the different data characteristics.

In other words, LSH is able to preserve a high probability P of hashing two data into the same hash bucket with hash function h if they are originally similar as raw data, or more formally: $sim(X,Y) = P\{h(x) = h(y)\}$.

2.3 Husky

Even with the fast growing computation power, we are still constrained by limited amount of memory if our program can only be commanded on a single machine. Therefore, to enable flexibility and scalability, the importance of a distributed environment has significantly increased. Husky, the open source platform, is an open-source project that enables high control yet minimal programming effort when building distributed applications.[10] Husky enables users to define objects that can cooperate within master

and different worker machines via pull, push and migrating messages and data. The framework itself performs load balancing and fault tolerance while enabling synchronous and asynchronous computation when workers are handling objects operations. This architecture serves as an important standing point for our generalized LSH framework to stand and develop in a later stage.

3 Image retrieval framework using nearest neighbor

Our goal is to retrieve the most similar image from a large amount of image stocks when a user provides a query image that has been modified through scale or rotation transformations. We show the abstract of our aim in Figure 1.

Figure 1. Partial Image Retrieval

3.1 Feature extraction

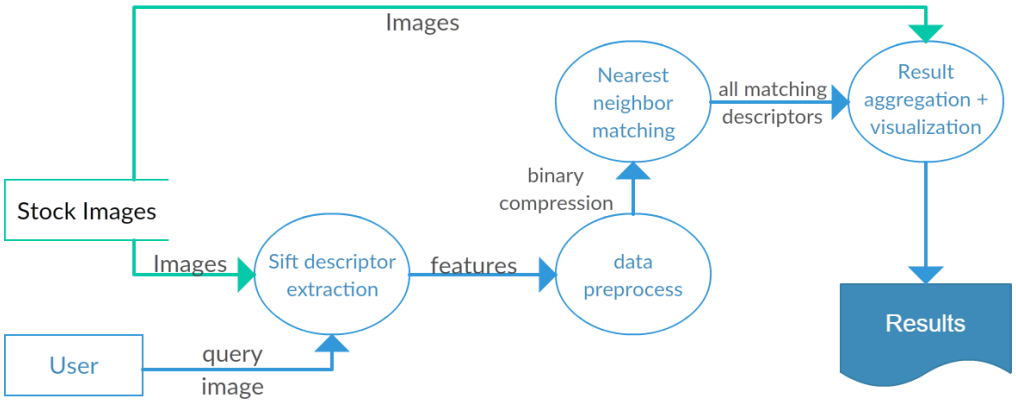
Feature extraction plays one of the most critical parts of machine learning process as it will deeply affect the training and classification of a particular problem. To begin our project, selecting the method to obtain a set of data points as representation of any given image is important. Scale-invariant feature transform (SIFT) is an algorithm proposed by David G. Lowe and has been widely adapted due to it's invariant features.[7] It looks for an image's local maxima in scale space of Laplacian of Gaussian (LoG) to identify points of interests. By tuning the σ of LoG, the algorithm is able to locate an image's feature point that uniquely identifies a characteristic of the image. The formula is defined as: $G(x, y, k\sigma)$. Once these key points are identified, the algorithm refines the points with Taylor series and narrow the results to high-contrast and smooth key points. With these key points, SIFT then magnifies the neighborhood pixels which would be represented in a 8 bin orientation histogram. Finally, a 128 dimensional vector is generated for a key point of the image.

The advantage of SIFT descriptor is that it is robust under the change of viewpoint perspectiv. In other words, the orientation and 3D perspective of an item does not affect the generated SIFT descriptors and the value is most likely preserved. The locality of these characteristics on the image is also recorded

in the descriptor. Last but not least, by providing large quantity of descriptors for each image, is becomes very handy to support our aim to perform image matching using nearest neighbor search. We can see an example of this descriptor to two images of the same object in the example below (Figure 2). Figure 2 consists of two images and their corresponding SIFT key points. The left image is the correct orientation of the dictionary but with lighting variation. The right image is a rotated version with correct lighting. SIFT is able to to extract almost identical key points (colored circle) in both images.

Figure 2. Demonstration of SIFT descriptor

Figure 3. Data flow diagram of IR system



3.2 Program design and Implementation

After confirming our use of feature descriptors, we now present our program design with the following data flow diagram (Figure 3). We modularized the entire program into three big sections: (1)

feature extraction using SIFT, (2) **nearest neighbor matching** and (3) **result aggregation and image retrieval**. In this section, we will focus on the first and third modular that focus on the extraction and feature matching from the results. We shall introduce our nearest neighbor matching method in the following two sections.

Feature extraction using SIFT supports multiple or single image data extraction. Our program not only extracts a given image to SIFT descriptors, but it also has the ability to write the stock images from the entire directory to a file of SIFT descriptors. Both of these operations will not save the SIFT data as txt files but instead as binary files. At the same time, for easy retrieval later, the program also renames the files to create easy-to-recognize Image Ids.

Since each image will produce numerous SIFT descriptors, a unique identifier has to be implemented. We design the Keys for each of the SIFT descriptors to be a combination of SIFTID+FileID, where SIFTID is identical within the image itself and FileID, it ensures the key's uniqueness across the directory. The algorithm details are listed as follows.

Algorithm 1: Image feature extraction

Input: image I or image directory and number of images
Output: binary file of sift vectors

```
initialization;
for image in images do
    extract keyPoints of given image(s);
    calculate descriptors from keyPoints;
    for descriptor of each point do
        concatenate key, descriptor, imageId;
        write as binary to output file;
    end
    rename images for results aggregation;
end
```

On the other hand, for **result aggregation and image retrieval**, we designed our algorithm detailed in Algorithm 1. Given that there are various SIFT key points from our query image, we provide an unique ID, or queryKey. The same thing applies to the stock images, the resultsKey. Our **nearest neighbor matching** outputs the possible candidates of nearest neighbor points in triples formatted as (queryKey, resultKey, actualDistance). These results are the nearest neighbors of the given queryKey found from the set of resultsKey. ActualDistance refers to the distance of some given metric space between the two keys. With this output, the following algorithm finds the image Id that associates the maximum amount of matching points. It would then retrieve the image from the stock. Finally it outputs an images that

shows the matching SIFT descriptors. The demonstration is detailed in section 8.

Algorithm 2: Result aggregation and image matching

Input: ResultFile, QueryImage, SearchDirectory

Output: matching images

```
initialization;
for each matching pair do
    extract fileIds from the results;
    map the result to FileId map;
end
extract all matching points of the max(FileId);
retrieve the image from directory;
matchKeyPoints of queryImage toward retrievedImage;
```

3.3 Nearest Neighbor Matching

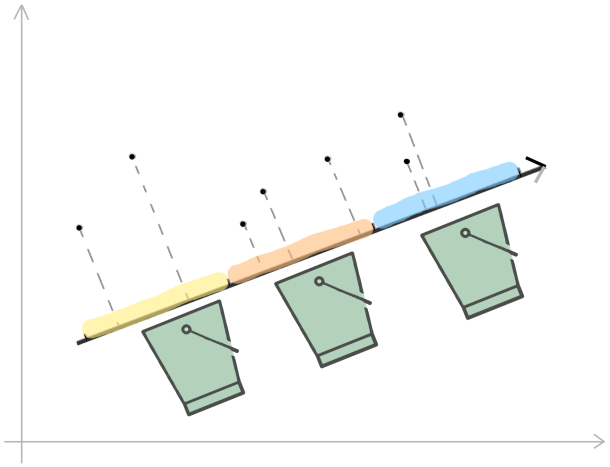
The selection of Nearest Neighbor matching algorithm for similar image matching becomes the most crucial piece of the puzzle. We must employ an effective and efficient nearest neighbor search method. When it comes to nearest neighbor search, kd-tree is one of the Best-known algorithms. It is a binary tree designed to recursively split the data into segments in the given metric space by dimensional[3]. What this implies is that for every tree node, it depends on the variance of a specific dimension until each tree node contains only one single entry. The reason why a kd-tree could be efficient is because the construction of the tree allows some degree of separation of distance between the nodes. While conducing searching, the process of traversing through tree nodes allows pruning. It tends to eliminate some of the unnecessary paths to look for nearest neighbors.

However, from a high dimensional perspective, the distance between nodes are consisted of a distance contributed by all d dimensions of the data such that a single node on kd-tree is not powerful enough to decide if the sub-tree underneath could be pruned. This falls back to a similar computation complexity as linear search or even worse[11]. Therefore, we decide to make use of Locality sensitive hashing to perform the nearest neighbor matching.

4 E2LSH Implementation on Single Machine

Given that the SIFT results are in real number vectors, we decide to use Euclidean metric space to find our nearest neighbor. An Euclidean distance can be defined as: $d(p,q) = \sqrt{\Sigma(q_i - p_i)^2}$. This method is widely known as the E2LSH proposed by Alexandr Andoni and Piotr Indyk [4, 11]. The intuition is to project the given high dimensional vector input onto a real number line in a given hyperspace. These points will then be partitioned into equal-width segments that can be considered as buckets (shown in figure 4).

Figure 4. Intuition of E2LSH



Algorithm 3: E2LSH

Input: QueryData q, Data D

Output: nearest neighbor

```
initialization;
readData();
createFunctions();
create hashTable T;
for each hashFunction h do
    for each data d in Data do
        Calculate the signature by h(d);
        store in T;
    end
    Calculate the signature by h(q) ;
    store in T;
end
Max = ∞ ;
for each row in T do
    if value equals h(q) then
        mark the data d as candidate;
    end
    calculate the nearest neighbor;
    replace the nearest neighbor if distance smaller than MAX
end
return MAX and the associate data d;
```

Since LSH requires some function family that preserves the data set's locality, Alexandr Andoni and Piotr Indyk introduced a hash function with p-stable distribution [4]. The function can be given as $a \cdot v + b$ which can be easily related to the project mentioned above. Here, it was explained that vector a is chosen from a normal distribution of number within 0 and 1 and the offset b is chosen from a uniform distribution of the proposed width for each segment. To retrieve the segment of the given projection, the hash function is formally defined as the following:

$$h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{w} \rfloor \tag{1}$$

where a is a uniformly generated vector with same dimension as v and b is a uniformly generated bias. We can see that similar data should be projected closer given the projection parameter is the same. Here we can describe our implementation of E2LSH on a single machine in algorithm 3.

Here we can see that a single E2LSH creates h hash tables that are d by h in dimension where d is the number of data in data set D and h is the number of hash function generated. Even if the distribution maintains the originality for more similar items to be hashed together, the randomness does not significantly increase the variance of such probability.

Therefore, an additional bands by rows design is implemented. What that means is implementing AND, OR operations toward the hash values. The OR operation is similar to the original setting where each data d that are hashed to the same hash value with the query are considered as candidate pairs. By incorporating several hash functions toward a single band, the possibility of a data d being hashed to a band of bucket b decreases, thus enhancing the probability of more similar items to be hashed to the same hash value, we define this as an AND operation. Through AND and OR operations we can magnify the variance of data projection such that the more similar items, the higher the probability and vice versa.

5 Distributing E2LSH

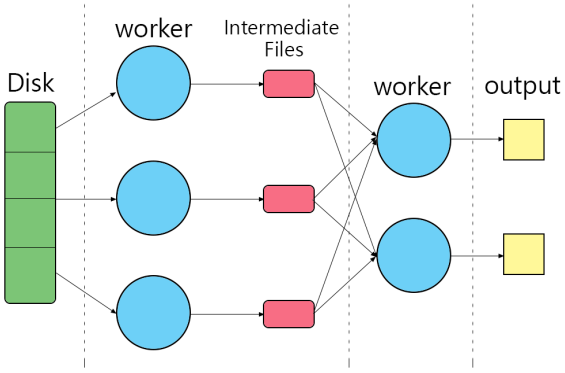
With the above E2LSH implemented, we can already prove our concept in achieving simple image retrieval on a single machine. However, our aim is to efficiently look for similar images within large data set of stock images that could easily exceed the memory of a single machine during image retrieval. Therefore, we need to alter our the algorithm so that it fits a distributed environment using the husky framework.

5.1 A Map-Reduce Framework

On Husky, MapReduce is one of the well supported programming frameworks in distributed environments. The framework proposed by Jeffrey Dean and Sanjay Ghemawat can be described with the following idea and execution graph (Figure 5)[12]. The input files are distributed across the machines and are stored evenly across the different workers. These workers would read the data from the distribution to perform some kind of task. We call this the map stage. These tasks are then to be processed

and calculated based on user defined logic. These programs write intermediate files locally. Intermediate files would then be remotely read by the workers again as data inputs, but this time they would move forward of the program and perform a new task such as aggregating the results. We call this the shuffle stage where the intermediate data moves toward the worker that is designed to handle similar types of intermediate data. Finally, the step of reduce where the workers would aggregate the computation and write the file output. There could be several shuffle stages in between the map and reduce stage. This framework allows data to be processed asynchronously in parts but synchronously as a whole.

Figure 5. Map-Reduce framework



As Husky implements it's framework, it maintains the Master Worker architecture by constantly creating list_execute as a queue of operations. The utilization of push messages channels enables smooth flows of message and intermediate results to pass onto the next round. The final output aggregation lies back into the Master's hands where it would return the results to the user. This is described to be compatible and easily reproducible into a MapReduce framework, or the MapReduce Chain[10].

5.2 Design and implementation

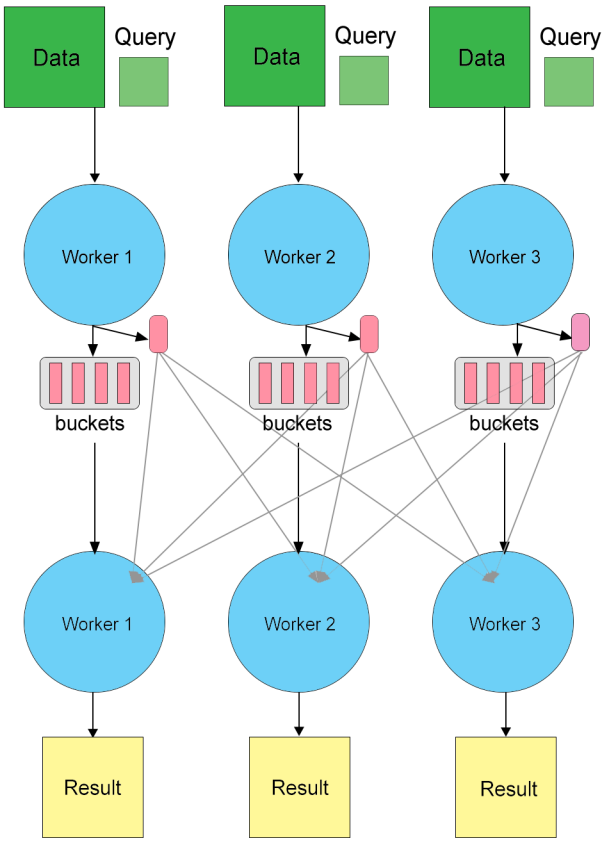
To transform a single machine E2LSH onto a distributed environment, we believe that MapReduce is a good starting point. From the previous section, it is clearly seen that the challenge here boils down to the large hash table that has to be maintained. We can see this hash table from Algorithm 3. To tackle this problem, we try to utilize the push messaging channel on Husky. Knowing that the number of queries is often far less than the subject data set itself, we decide to conduct the distributed LSH shown in the following manner (Figure 6).

What we want to achieve here is to create a sparse hash table, instead of a dense one, such that we can distribute the hash table across the board. From the top section of the figure 6, we can see that a proportion of both the queries and items should be distributed across the machines. Instead of keeping a hash table on each machine or across the board for update, we utilize the object-oriented programming

paradigm provided by husky and create the so called bucket objects. These objects would have a unique id as key. Each worker would asynchronously follow the following steps:

1. Retrieve the random seed from master to generate the different a and b for hash function associated with the previous E2LSH formula.
2. Calculate the hash values of given proportion of data given the aforementioned formula.
3. Create bucket associated with the hash values.
4. Calculate the hash value of the proportion of query data.

Figure 6. Distributed E2LSH



Different from the single machine algorithm, we do not aggregate any hash table on different machines. Rather, we keep the buckets created by the data locally but make use of the map-reduce framework – pushing intermediate messages to other workers. Here we will send the message of the hash value (the pink-rounded box) of the queries across to different worker machines (demonstrated as a small box beside the buckets), or shuffling. Since queries are usually far less than the amount of data it has to search from, it minimizes network traffic and enhances the performance.

From here we continue our journey where workers would pick up the hash value of given query and compare the query toward the buckets already created on it's local machine. Finally it returns the minimum distance of the query and data found within the same bucket if it happens to exist.

6 A Generalized Framework for Distributed LSH

Given the implementation of distributed E2LSH, it is meaningful to take a step back to revisit the LSH algorithm. Locality sensitive hashing algorithms tend to follow the following algorithmic steps:

1. Create some hash function according the the given distance metrics setting and other relevant parameters given by the user.
2. Hash the data and query toward function created in step 1 and create a hash table.
3. Within the same hash table of the hash function, retrieve the data and query that are hashed together and perform a linear search to identify the nearest neighbor.

Variations of different LSH algorithms are often due to the use of different metric spaces from step 1 and the different method to identify and locate the actual nearest neighbor mentioned in step 2. For example, Hamming distance LSH make use of hamming distances and K-means LSH utilizes clustering techniques to provide the bucket IDs. Muti-probing LSH, on the other hand, changes the results phase where it does not halt until a nearest neighbor is found. If we see this from a broader term, instead of buckets, we are creating indexing collision and answering the query given the index. Therefore, it is very likely for us to create a general framework for the LSH algorithm that can allow users to easily change the LSH Hash family to suit their need providing the different data sets.

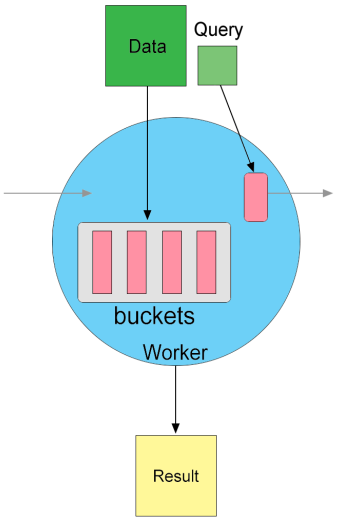
6.1 Introducing the abstraction

If we abstract Figure 6 from our idea to build a distributed version of E2LSH, we can construct a generalized framework of Distributed LSH as Figure 7 consisting of three major phases: "query-forwarding", "bucketing and indexing" and "query-answering".

If we only draw one single worker, we can see that the flow now contains both the data and query entries distributed on the worker machine. It would go through the "query-forwarding" phase (the small green boxes toward the small pink rounded-box). Here, each worker machine would initially develop their own hash functions based on user given settings. Query data will utilize these hash functions to calculate the associated hash values. The values will be forwarded toward other machines seen by the gray out-going arrow. At the same time, data on the machine creates buckets using the same hash functions used by the query to generate various hash-buckets. This can be seen by the larger green box drawn towards the buckets. These buckets would act as intermediate data and will be stored locally. This is defined as "bucketing and indexing".

The ”query-answering” is described as the incoming gray arrow passing through the buckets and generating the results. The worker picks up the broadcast query hash values from the query forwarding phase and looks toward the bucket and examine if any of the data locally falls into this particular bucket. If this bucket does exist, the data in the bucket(s) would then be evaluated by calculating the exact distances and answers the query. The system would halt here for typical LSH searches. However, if this LSH has any type of probing algorithm, it can create another set of queries and broadcast the information to initiate another round of the work flow. Here we can summarize that our general distributed LSH framework consists of ”query-forwarding”, ”bucketing and indexing” and ”query-answering” phases.

Figure 7. General Distributed LSH framework



7 Implementation of other LSH

Here we demonstrate the implementation of different LSH algorithms on this general framework.

7.1 HLSH

Hamming Distance LSH(HLSH) relies on bit-sampling when conducting similarity searches [8]. The name suggests that the given distance metrics is in hamming distance formally defined as:

Definition 3: Hamming Distance

$D = \Sigma \mid x_i - y_i \mid$ *where if* $x = y \Rightarrow D = 0$ *and* $x \neq y \Rightarrow D = 0$

When given a binary vector $v \in \{0,1\}^d$, Hamming distance LSH randomly samples $k \in \{1,2,,d\}$ from the d dimensions of the given data set such that $h(v) = y_k$. With the same concept as of E2LSH, k hash functions should be applied to any given point x and then concatenated together, denoted as $g(x) = (h1(x), h2(x), hk(x))$ as the AND operation mentioned in section 4. $G(x)$ would then denote the

signature for the bucket. However, this would reduce the chance of collision between similar items which is why L signatures has to be generated, the OR operation mentioned previously. The general purpose of doing so is to assure a higher chance for similar items to be hashed together and vice versa.

Following the abstracted framework mentioned in the previous section, we now explain our implementation of HLSH. When a data vector is presented with an Id and a vector of $\{0,1\}^d$, we split them to different workers across the distributed system. We allow the format of the vectors in both dense or sparse format.

We then create our query-forwarding step: Again, this step consists of creating the hash function as well as forwarding the query data. The hash function is described as pseudo-code as follows (Algorithm 4 and 5). Worker machines follow these algorithms to apply bit selection given data. This hash function would randomly select r indexes predefined by the user with a given seed. The hash function would then merge these indexes into a vector and represent them as a signature. This signature is the result of the second step "bucketing and indexing". The two algorithms only differ by the way they interpret given data while both will output a set of $\{0,1\}^n$ vectors demonstrating the selected bits.

To avoid unnecessary data traffic, we allow each machine to have a copy of the query stored within the machine in the first place. The workers will evenly process q/w queries, where q is the number of total queries and w is the total numbers of workers, to obtain the $\{0,1\}^n$ signature of the identical queries. These signatures would then be broadcasted across the network and then each worker would use the received signature and find the data within the bucket stored on that particular worker machine.

Finally, in the "query-answering" phase, the workers would also process the data in the bucket and calculate the exact distance of the given query. Each of these results would then be sorted using `std::sort` and the top k elements would be extracted.

A single bit-wise hash function might not allow us to truly widen the probability for similar objects to be hashed together and dissimilar objects to be separated. Thus, we would again use the AND-OR composition of hash functions mentioned in section 4 to enhance the performance of HLSH's hash function. Here we also distribute this method in an distributed manner without maintaining a full hash table across the board. We can imagine our generation of series of index into a vector as signature would be the AND operation. It concatenates the selected bits together. As each bit can either be 1 or 0, the more concatenation, the more possibility of bucket ID can exist. Thus, it narrows the number of data that could possible to be within a particular bucket. The OR operation can be seen as allowing the same data to be hashed to multiple buckets, thus increasing its chance to collide with other data sets instead of only one single bucket. Therefore, during the implementation, we design our algorithm to not only sample the n number of bits the user defined, but also b sets of these signatures. Each of these signatures can be viewed as a separate hash function. Thus for every signature we would append the index of the band toward the end of the signature as the finalized bucket ID. The whole hash process is shown in Figure 8.

Algorithm 4: Sparse Vector Hamming Function

Input: SparseVector V, numberOfFunction n, numberOfCompoundFunctions c

Output: vector of bucketIDs

```
initialization;
generate nc bits_to_sample;
for b in bits_to_sample do
    if b in V then mark 1;
    else mark 0;
end
foreach c of n do  concatenate results ;
send vector of bucket IDs
```

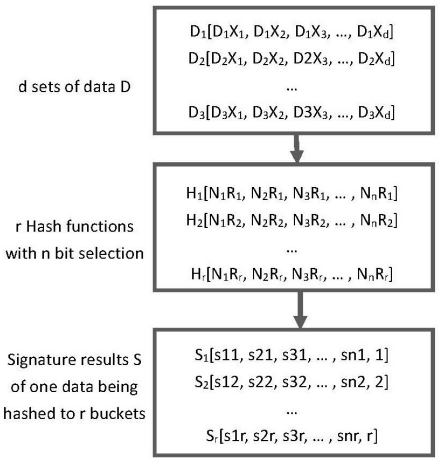
Algorithm 5: Dense Vector Hamming Function

Input: DenseVector V, numberOfFunction n, numberOfCompoundFunctions c

Output: vector of bucketIDs

```
initialization;
generate nc bits_to_sample;
for b in bits_to_sample do
    save value of V[b];
end
foreach c of n do  concatenate results ;
send vector of bucket IDs
```

Figure 8. Hamming Distance LSH Hash flow



Other enhancements for performance can be done for HLSH. One of the significant issues in mem-

ory allocation is the lengthy signatures. Signatures are essential to identify buckets and thus each data set, both query or training data, would have several signatures, or bucket ID in our system. It not only causes time delay for comparison but also leads to significant memory storage. For example, a bit selection of 32 dimensions as signature would require 128 byte per bucket, resulting in some overhead.

Therefore we present a special processing of the hashed signature before saving it into the data. Using the understanding of the availability to represent signed-integer in 32-bit binary format, we allow each bucket id, originally presented as $0,1^n$ into 32 bits. We fill the empty spaces with 0 if the user did not select more than 32 dimensions as their bit-wise parameter for each concatenation. We then apply bitwise operations within C++ and save them into a int (or several ints). This way we are able to represent a unique binary bucket ID and transformation it using a 4 byte unsigned-int as the final signature.[13]

7.2 C2LSH

Collision Counting LSH (C2LSH) modifies the way the distributed LSH handles "query answering" instead of changing the distance metrics.[14]. C2LSH extends E2LSH by allowing dynamic change to the w noted in equation 1. Knowing that E2LSH aims to hash data points by projecting closer points in Euclidean space into the same bucket, it is intuitive that the next to closest ones should be in nearby buckets. With this idea, C2LSH aims at probing buckets that are closer to retrieving a user defined k-nearest neighbors.

To accomplish such task, the hash function is modified to the following family where R is an integer power of c and $R \leq c^{\lceil log_c td \rceil}$ and the rest follows E2LSH:

$$h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{wR} \rfloor \tag{2}$$

and therefore in each iteration, value R will increase by a magnitude of power widening the "width" of the buckets. Thus, we modify "query-answering" to first check if the bucket the query is sent to fulfills the k elements that the user requested for. If not, we would apply the new hash function toward the query and send the corresponding parameters to enter the next iteration of LSH.[15]

8 Evaluation and Results

8.1 Image retrieval

Our data set consists of random high resolution images from free stock photo libraries from the internet. We would randomly select one of them to do a crop, rotation and minor lighting adjustments. We demonstrate several experiments as example and the metrics we measured. We first show small sets of data of books in the first experiment. Then we show our first pressure experiment in objects that are from the same category to see if it can still recognize the original image. Finally, we would perform an example on querying a huge data set.

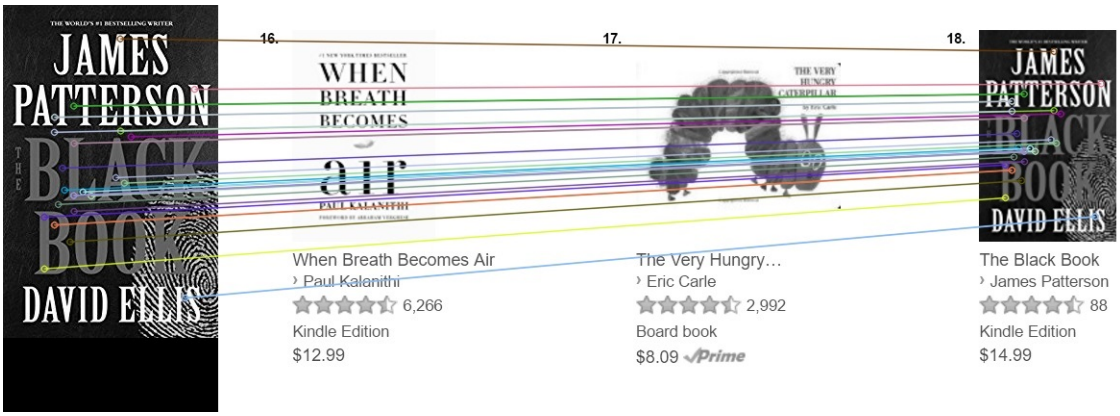
As SIFT point generation could be a one time process, we would only measure the time for nearest neighbor search in the following experiments.

8.1.1 Same item query

In this experiment, we screen captured 7 rows of items from Amazon.com. We then randomly selected one item from one of the rows as our query images. Here we did not crop that image directly from the row, but we searched for that book and found another image from Amazon. 18940 SIFT data points were created from these images with the binary file size of 9.4 MB. The total time spent with a single machine single thread setting is 0.488 seconds. The results is shown in Figure 9.

We merged the query and the retrieved image from the given directory to give the demonstration. The left image is our query image while the row of books on the right is the retrieved results. Here we can identify the matching points found on the left hand side to the right hand side. A total of 29 nearest neighbors were found.

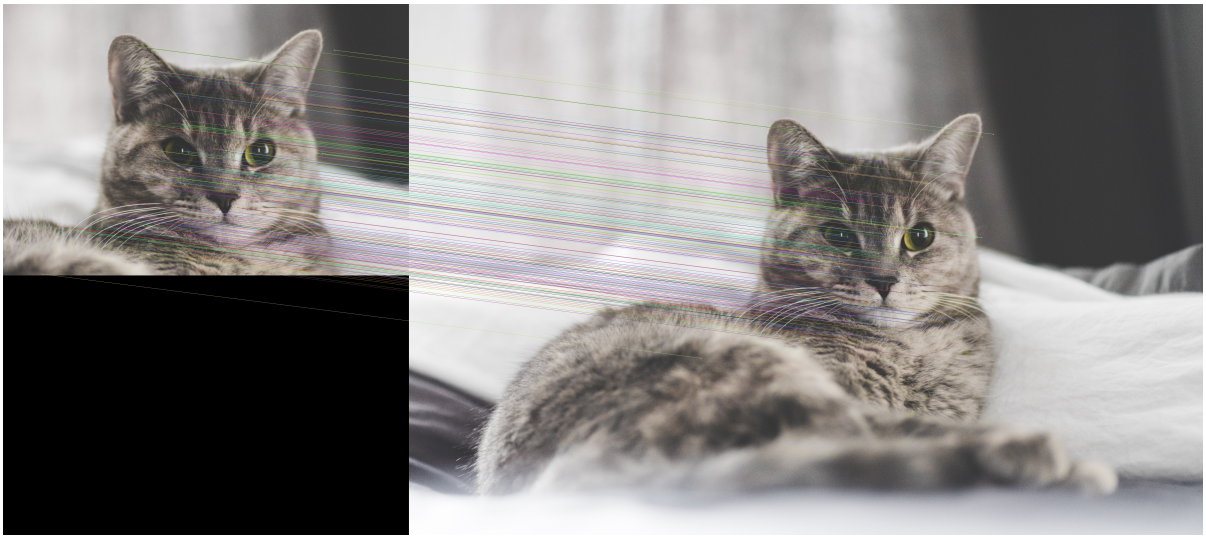
Figure 9. Experiment 1: Same item query on mini data set



8.1.2 Same category query

Knowing that we are now able to retrieve a book, we want to know if putting images of the same category would confuse the detection. We also increased the amount of query. This time we downloaded 30 high quality photos of cats with the total file size of 339 MB consisting of 680 SIFT data points. We cropped and altered one of the cat images as our query image. From the image below we can see that large amounts of SIFT points were located as nearest neighbors.

Figure 10. Experiment 2



8.1.3 Huge data set query

Finally, we want to perform a final testing for huge data set that contains images of mixed categories, sizes and file size. In figure 11, you can see a brief part of the high resolution stock images that range from 51MB to 41KB in the directory. We have 500 images as our query images that included **13.5 million** numbers of SIFT data points. Again, we cropped, rotated, and alternated one of the images as our input data. we can see that it successfully returned the image in figure 12.

Figure 11. Stock images

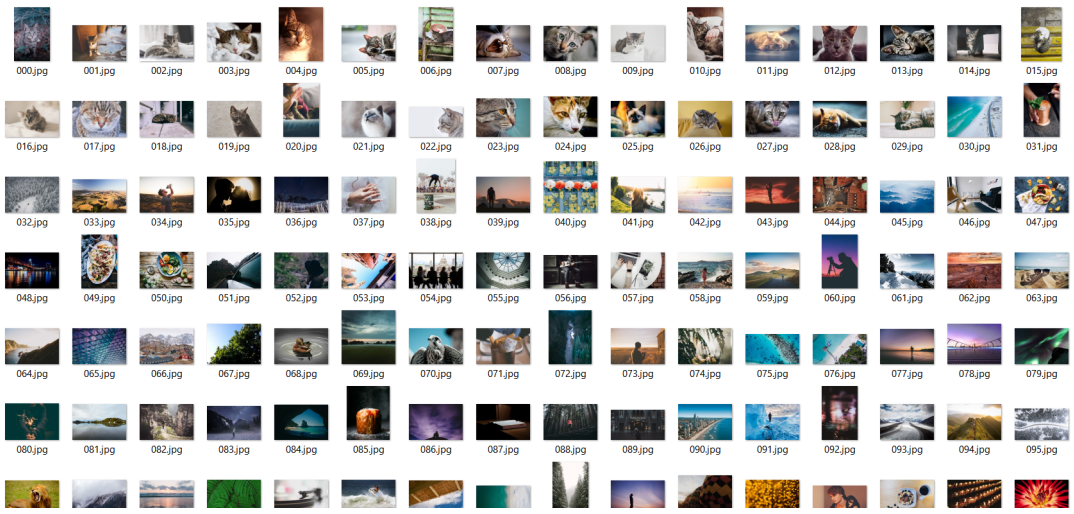


Figure 12. Huge data set matching

8.2 Comparing FLANN

Fast Approximate Nearest Neighbor Search(FLANN) is a library contributed to the well know computer vision open source project OpenCV for similar feature matching[16]. Created originally by Marius Muja, David G. Lowe, it is currently maintained and enhanced by the community. One of the greatest weaknesses of FLANN is the limitation of single machine computation. As the search methods are already incorporated into APIs (Application Programming Interface), it is difficult for users to extend and modify the search algorithm into a distributed manner. Thus, it is easy to imagine the issue when there are too many stock images and a single machine, which generally has limited memory, becoming incapable to retrieve the correct image.

In this part of the experiment, we tested 130 images consisting of 4 million data points against a query image of 8551 data points. We can already see a clear difference in time on a single machine single thread setting. Comparing it to allowing parallel computation by husky, the time shortens significantly.

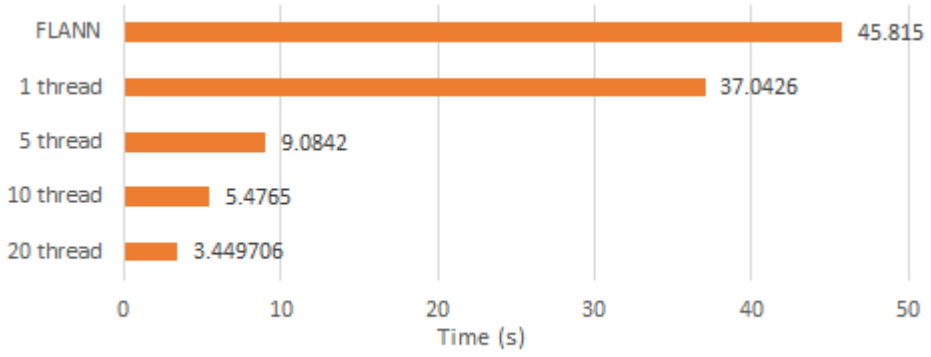


Chart 1. FLANN compare

9 Discussion and Conclusion

Form the results, we can clearly see that we have successfully reached our goal in image retrieval. From the program design to distributed computing, we are able to retrieve the target image within a short amount of time with high accuracy. By using the SIFT descriptor, we were able to maintain the key features of any given image despite the change in scale, orientation and various lighting adjustments. Through the use of LSH, especially through distributing the LSH, we are able to demonstrate efficient nearest neighbor search with the use of large and high dimensional data sets.

With the support from the Husky framework, we are able to build a distributed E2LSH. The introduction of generalization of LSH into a distributed environment is the key to create many other usages of distributed LSH algorithms. With more distance metrics, wewill not be limited to image retrieval but many other use cases.

9.1 Issues and weakness of the IR system

However, it is still worth mentioning that from the experiment, one of the weaknesses of our image retrieving program is the difficulty to find a generalized object, in other words a narrow classification. For instance, identifying the same person across multiple images or the same kind of food.

The key reason goes back to our feature extraction method, SIFT, that is most suitable to maintain the originality when a image is scaled and tuned. By this means, objects, such as books and products, are better performed in our system. For example, the same people in different images could have performed a different gesture and expression. This could strongly affect the borderlines and the key feature points spotted by SIFT in the first place.

The other issue observed with the construction of this program highlights the physical meaning of distance metrics. Given a SIFT feature of 128 dimensions, it is difficult to justify the use of Euclidean distance instead of other measures, such as Euclidean Squared distances which might highly enhance the importance of a the greater value compared to the smaller.

9.2 Possible improvements for the IR system

With the above mentioned issues, it is clear that we have some improvements that could be completed. The user interface of the system will be open-sourced for future improvements. Several other distance metrics and feature extraction methods can be included for variation of nearest neighbor detection. For example with the use of SURF descriptors, we might be able to decrease the amount of processing time using the same set of images.

9.3 Future Plans

Focusing on implementing more general LSH framework is important. Useful distributed LSH includes k-means LSH[17] that utilizes the powerful k-means clustering technique to act as the indexing

heuristic for LSH search as well as a possible implementation of other LSH methods based on different probing techniques and metric distances. By incorporating more feature extraction methods as well as distributed LSH applications, we can alter between the techniques when searching different images.

10 Acknowledgments

This Final Year Project has been extremely fruitful. Standing at the crossroad between distributed computing, machine learning, computer vision and data science has been challenging but enjoyable. Without prior knowledge in the four fields, I am excited to learn along the way while getting my hands dirty, implementing different ideas and examining the hypothesis.

With that being said, I am very grateful to have the opportunity to soar with the eagles, test the theoretical and apply the practical. I am extremely thankful to my adviser, Professor James Cheng, who gave me the opportunity and time to learn and grow. I must also thank my tutor Mr. Jinfeng Li for his knowledge, patience and guidance.

References

[1] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. Aug 13, 2014.

[2] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. A survey on nearest neighbor search methods. *International Journal of Computer Applications*, 95(25), Jan 1, 2014.

[3] Jon Bentley. Multidimensional binary search trees used for associative searching, Sep 1, 1975.

[4] Alexandr Andoni and Piotr Indyk. E2lsh 0.1 user manual.

[5] Wang Weihong and Wang Song. A scalable content-based image retrieval scheme using locality-sensitive hashing. volume 1, pages 151–154, 2009.

[6] Hsin-Liang Chen and Edie M Rasmussen. Intellectual access to images. *Library Trends*, 48(2):291, Oct 1, 1999.

[7] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004.

[8] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. page 604613, New York, NY, USA, 1998. ACM.

[9] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Semi-supervised hashing for scalable image retrieval. pages 3424–3431, 2010.

[10] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proc. VLDB Endow.*, 9(5):420431, January 2016.

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. SCG '04, pages 253–262. ACM, Jun 8, 2004.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce, Jan 1, 2008.

[13] Ti-Chung Cheng. Efficient nearest neighbor search in distributed manner. Final year project midterm report., dec 2016.

[14] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. Locality-sensitive hashing scheme based on dynamic collision counting. SIGMOD '12, pages 541–552. ACM, May 20, 2012.

[15] Ti-Chung Cheng. Implementation and analysis of collision counting lsh on the husky framework. Distributed C2LSH on husky for CUHK 2016 summer research project, aug 2016.

[16] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[17] Eliezer S. Silva and Eduardo Valle. K-medoids lsh: a new locality sensitive hashing in general metric space, 2013.