Building KNN Graph for Billion High Dimensional Vectors Efficiently

Yue Leng, Xinxun Zeng, Zhenyu Chen

1 INTRODUCTION

KNN refers to "K Nearest Neighbors", which is a basic and popular topic in data mining and machine learning areas. The KNN graph is a graph in which two vertices p and q are connected by an edge, if the distance between p and q is among the K-th smallest distances.[2] Given different similarity measure of these vectors, the pairwise distance can be Hamming distance, Cosine distance, Euclidean distance and so on. We take Euclidean distance as the way to measure similarity between vectors in this paper. The KNN Graph data structure has many advantages in data mining. For example, for a billion-level dataset, prebuilding a KNN graph offline as an index is much better than doing KNN search online many times.

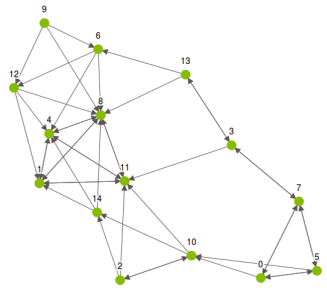

Figure 1: A sample of three nearest neighbors graph

The study case of this work is as follows: Alibaba has more than one billion products in Taobao platform, and each product contains many features: product name, image, description, tags and so on. For each product, it has already been represented to a float vector with 1024 dimensions by learning algorithm. There are two main challenges on this situation: (1) how to construct a KNN graph quickly, and (2) how to answer a KNN query, which means given a query point, search in the KNN graph index to find its k nearest neighbors as accurate and fast as possible. In this work, we mainly focus on the first challenge, which aims to propose an algorithm to build the KNN graph for the billion-scale products in Taobao efficiently.

2 RELATED WORK

Nearest neighbor search has been a very popular research topic during the last decades. However, as the scale of dataset increasing rapidly, it is almost impossible to find the exact k nearest neighbor in high dimensions Euclidean space. Therefore, researchers tend to focus on approximate nearest neighbor (ANN) problem, which can be performed efficiently and are sufficiently useful for many practical problems.

There are a large number of papers published on this research problem. Obviously, the naive approach to construct a KNN graph is repetitively applying KNN search for each vector in the whole dataset, and the time complexity is $O(n^2d)$. Therefore, in order to have a further understanding of this topic, we must consider the related work not only in approximate KNN graph construction, but also in approximate KNN search.

The traditional method of KNN search can be divided into two categories: tree-based method and hashing-based method. In addition, as the data structure KNN graph raise, the graph-based method, which can be used to do both

KNN search and construct KNN graph was put forward. Each method has their own advantages and drawbacks. In this section, we will introduce the tree-based method, hashing-based method and graph-based method briefly.

2.1 Tree-Based Method

Tree-based method is very popular in the initial stage of this topic. It gives a high recall query result in a small dataset, and even can be used for the exact KNN problem. This kind of method uses a tree structure to partition the whole vector space hierarchically, so that the vectors that be partitioned into a same node have high similarity between them. There are various trees based on the different partition rules and other limitations, for example, the kd-tree, k-means tree, VP-tree[13] and so on. FLANN[12] is a represented tree-based method, widely used in recommendation system. Given a dataset, FLANN automatically choose the most suitable one within randomize kd-tree, hierarchical k-means tree and linear scan by minimize a specific cost function.

However, the tree-based method suffers from the curse of dimensionality. The cost of time and space is unacceptable for high dimension and large-scale data when using tree-based method.

2.2 Hashing-Based Method

This kind of method find nearest neighbors by applying hashing function to the original vector space, and then consider points in the same hashing bucket has higher possibility to be nearest neighbors of each other. The hashing-based method costs much less time and memory than tree-based method, especially in large-scale high dimension dataset. However, the recall of query result by hashing is the bottleneck.

One of the most popular hashing algorithm in KNN is locality sensitive hashing (LSH), which was first introduced by IndyK and Motwani[9] in 1998. LSH-based method uses a family of LSH functions to reduce the dimensionality of high dimension data. The idea is that the closer p, q in original space, the more likely p and q collide after hashing. The LSH-based method in the early stage use random projection as hashing function, and defined in Hamming distance[7]. However, with the development of LSH, there are many variants now, such as C2LSH[6], E2LSH[1] and QALSH[8].

2.3 Graph-Based Method

The different between graph-based method and other method is it use KNN graph as an index, so that it can benefits from some features of graph index. One of the earliest efficient method to build approximate KNN graph with arbitrary similarity measure is KGraph[3], also known as NN descent, published in WWW conference 2011. The method is based on a simple principle: a neighbor's neighbors is also likely to be a neighbor. Therefore, this method randomly choose k neighbors of each point at first, and then by exploring neighbors' neighbors to choose the k nearest ones of each points iteratively, the KNN graph converge and become more accurate. Compared with the brute force algorithm of building KNN graph, KGraph does not need to scan the whole dataset to find the nearest neighbor, which greatly reduce the time.

Some algorithms to improve KGraph were raised these year. EFANNA[4] applies kd-tree as the way to initial k nearest neighbors of each points instead of randomly selection, which speeds up the process of NN descent. In addition, some work such as NSG[5], try to design a better graph structure to increase the performance.

3 METHODOLOGY

A novel approach named KGraph proposed by Wei Dong et al. at 2011 [3] is basic framework for KNN Graph Construction problem. This algorithm is simple but efficient. We take it as the baseline. After the raised of KGraph, more and more graph-based method based on the idea of KGraph were put forward. To the best of our knowledge, EFANNA[4] is one of the state of the art to build a KNN graph with the high recall. In this section, we will first introduce this KGraph algorithm in detail, and then state the idea of our approach to build a KNN graph efficiently.

3.1 Basic Idea

The KGraph algorithm is based on a simple idea: neighbors' neighbor is likely to be a neighbor. That is, when you have some neighbors who are close to you, your neighbors' neighbor is likely to close to you too. In brute force algorithm, we scan all vertices to find out k nearest neighbor. In KGraph approach, you only need to scan your neighbors' neighbor, to find out who is closer to you and update your neighbor set. After several iterations, when you find there are no new neighbor can be updated, the algorithm stops. There are one problem remains: how to find your first neighbors. In the KGraph [3], random selection is okay.

Our goal is to make some improvements based on the KGraph algorithm in two aspects: First, start the algorithm with an approximate KNN graph instead of random initialize graph. An approximate KNN can be generated by many methods, for example, the kd-tree or LSH algorithm, which takes an acceptable extra time but get much more accurate result of initialization. Good initialization can reduce the times of iteration, which can save time further. Second, we want to optimize the distance computation process in NN-descent iterations. In each iteration, each vector need to check all its neighbors' neighbor, and compute distance with them, which is $O(n^2d)$ complexity. In a billion-level dataset, the time cost in distance computation becomes the bottleneck. It is better to find a way using lower cost to achieve the same result. We list some optimization to speed up the distance computation cost in optimization part.

3.2 Notation

Let V be a set of vectors. For each v in V, let $N_K(v)$ be v's K-NN, i.e. the K vectors in V are closest to v, and also K neighbors v is pointing to in our KNN graph. Similarly, let $R_K(v)$ be v's reverse K-NN, which are pointing to v in graph. In algorithm, we take $T_K(v) = N_K(v) \cup R_K(v)$ as v's general neighbors.

3.3 Basic Algorithm

Step 1 Apply locality sensitive hashing function to each vector $v \in V$, get the hash table. Repeat this step several times by randomly choose hashing function form the hashing family, to get several hash table.

Step 2 Take each bucket in the hash table as a neighborhood group, let all the vectors in a bucket compute pair-to-pair distance with each other, and find the k nearest neighbors for them in their own buckets.

Step 3 By traverse all the buckets, we find the initial k nearest neighbors for each vector, and get an approximate KNN graph.

Step 4 For each $v \in V$, gather v's $T_K(v)$, for each $t \in T_K(v)$, for each $u \in T_K(t)$, calculate the distance of v and u, and then compare it with the original K nearest vectors of v and u. If the new distances is smaller, update $N_K(v)$.

Step 5 Repeat step 4 until there are no more change in graph or the termination threshold is reached.

The pseudocode of algorithms in the following page.

3.4 Optimization

3.4.1 Changeable length code of LSH.

The random project LSH has a critical drawback when finding k nearest neighbors in some specific data set: the distribution of vectors in each buckets is extremely uneven. Therefore, most of vectors are allocated in the same buckets because they are close to each other in the original space, while some buckets only have few vectors in it. This situation reduce the quality of initial KNN graph, and also cost lots of time in distance computation in one large size bucket. Therefore, we put forward a changeable length code of LSH to divide vectors into hash bucket recursively, which guides the even distribution of hash bucket.

3.4.2 The state flag for vector.

As algorithm runs, fewer and fewer new neighbors can update the graph, so we can set a flag to detect whether a vector need to recalculate its neighbor set to avoid unnecessary calculation.

Algorithm 1 CLSH - initialize KNN graph

Input: data set D, number of nearest neighbors k, number of table t, size of bucket b
Output: number of t hash table set s, initial KNN graph G_{init}

```
for i = 0 → t do
    CLSH(hashcode, 0, n)
end for

function CLSH(hashcode, left, right)
    if right − left < b then
        for i = left → right do
            node i join into bucket[hashcode]
            for j = i + 1 → right do
                calculate distance between i and j
                Add i to j's neighbors
                Add j to i's neighbors
            end for
        end for
    else
        for i = left → right do
            add one bit binary to hashcode of i by hash function
        end for
        divide = the position that separate all the nodes by 0 or 1
        CLSH(hashcode, left, divide)
        CLSH(hashcode, divide + 1, right)
    end if
end function
```

3.4.3 *Euclidean distance pruning.*

There are several ideas about how to save the time in distance computation. First, we can set a distance threshold according to the current nearest neighbors set, and then terminate the distance computation before computing all dimensions. Also, the triangle inequality can help us get the lower bound of a distance before exactly compute it. In addition, Transfer Euclidean distance to inner product is also a way to consider[10].

4 EXPERIMENT

To have a full comparison between our method and other types of algorithms designed to solve the k nearest neighbors problem, we apply our algorithms in two very popular dataset: sift and gist. And compare the result with the experiments published in a survey of ANN problem[11].

4.1 Dataset

We conduct our experiments in two popular real world data set for nearest neighbor problem. The detailed information are listed below.

name	dimensions	vector number	query number
SIFT	128	1,000,000	10,000
GIST	960	1,000,000	1,000

Algorithm 2 Refine KNN graph

Input: initial KNN graph G_{init}, data set D, maximum iteration number I
Output: approximate KNN graph G

```
for i = 0 → I do
    for v ∈ V do
        gather all v's neighbors and reverse neighbors into T_K(v)
        for t ∈ T_K(v) do
            gather all t's neighbors and reverse neighbors into T_K(t)
            for u ∈ T_K(t) do
                if u's flag is old  v's flag is old then
                    return
                end if
                dist = calculate the distance of v and u
                if dist<the k-th nearest distance of v then
                    add u into N_K(v)
                    set u state flag as new
                end if
                if dist<the k-th nearest distance of u then
                    add v into N_K(u)
                    set v state flag as new
                end if
            end for
        end for
```

4.2 Evaluation Metrics

We apply the same evaluation metrics with the 2016 survey paper[11].

For the efficiency, we use the **speedup** as metrics, speedup of an algorithm is t_{BF}/t, where t_{BF} is the brute force time of searching a query, and t is the average search time of a query by using the algorithm.

For the accuracy, we use the **recall**, $recall = |X \cap KNN(q)|/k$, where X is the set of neighbors given by the search algorithm, and $KNN(q)$ is the true nearest neighbors for query vector.

In addition, we also measure the **index construction time** and **index size** for KNN graph construction.

4.3 Compared with graph-based Method

The idea of our algorithm is based on the graph-based method. Therefore, we conduct some experiment in sift and gist dataset, to compare the efficiency and accuracy of our algorithm and two typical graph-based algorithms: KGraph and EFANNA. In this experiment, we set k=100, and the new neighbor checking number in each iteration is equals to k for the three algorithms. We measure the KNN graph construction time, and the average recall of the whole graph, The result is shown in Figure 2 and Figure 3.

In the result, KGraph is represented by yellow point, EFANNA is represent by blue point and our algorithm is the green one. We know that EFANNA and our algorithm is much better than KGraph, and our algorithm is comparable with the EFANNA is this two dataset. However, for much larger dataset, EFANNA will perform worse because tree-based intilization stage will suffer from the curse of dimensionality.

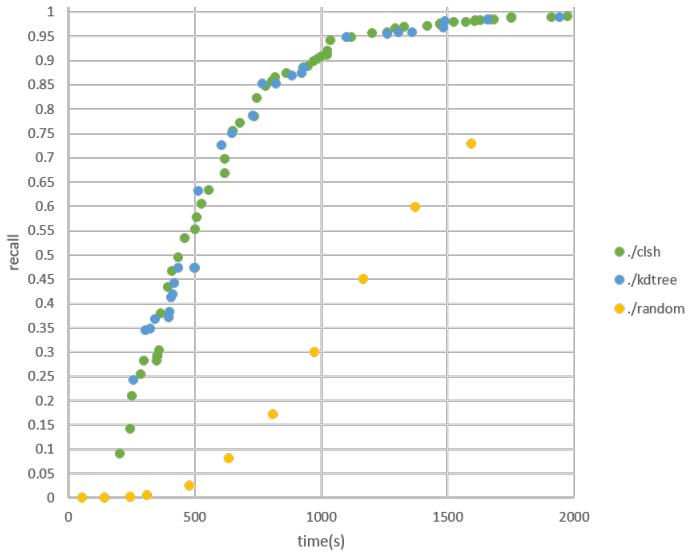

Figure 2: SIFT, time vs recall

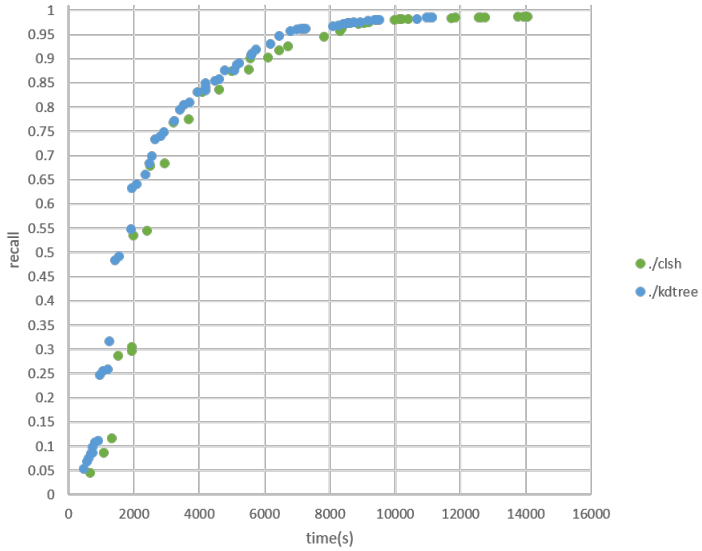

Figure 3: GIST, time vs recall

4.4 CLSH compared with learning-based Method

In this section, we only compare the CLSH, the graph initialization method, with the six data dependent learing-based hashing algorithms tested in the survey. We measure the relationship between speedup of one query and recall,and the result is shown in Figure 4. Our algorithm is shown in yellow line.

According to the result, we find that performance of our algorithm is quiet good, only post dominant by OPQ algorithm. However, the time used in index construction of OPQ is much bigger that our algorithm.

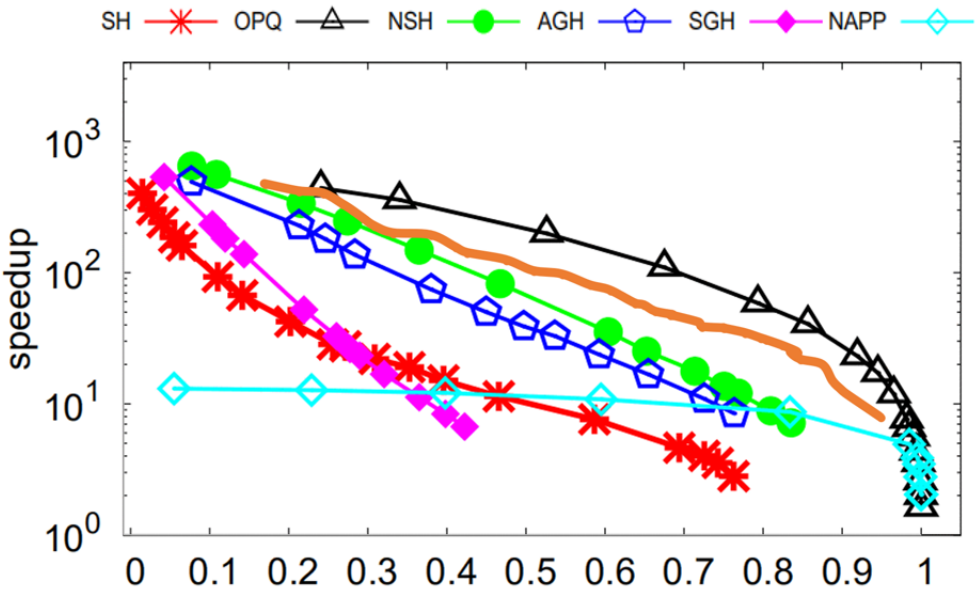

Figure 4: SIFT, speedup vs recall

	time (s)	size (MB)
SH	320	729
OPQ	788	65
NAPP	122	119
NSH	35	34
AGH	28	65
SGH	221	65
CLSH	30-300	40-400

5 CONCLUSION

In this semester, we get familiar with the KNN Graph construction and search problem, and also gave our own idea to deal with it. According to the experiment part, the performance of our algorithm is quiet good, but also have somewhere to improve.

Our work in this semester are divided into three stages:

1 Read 7 papers about KNN problem, get familiar with the topic. Read the source code of EFANNA(2016) thoroughly.

2 Read 5 papers about hashing and inner product. Try several ways to improve the cost of distance computation. In addition, we implement our hashing algorithm and combine with NN-decent. Conduct a series of experiments of LSH+NN-decent and KDtree+NN-decent.

3 Read 4 papers about hashing-based algorithm in KNN. Conduct experiments to compare our LSH method with other state-of-art hash algorithm. Draw conclusion and finish final report.

At last, we would like to thank professor Tang and professor Zhang, for the support and suggestion for the project.

REFERENCES

[1] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 459–468.

[2] Wikipedia contributors. 2017. Nearest neighbor graph — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Nearest_neighbor_graph&oldid=799649575 [Online; accessed 9-April-2018].

[3] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. ACM, 577–586.

[4] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).

[5] Cong Fu, Changxu Wang, and Deng Cai. 2017. Fast Approximate Nearest Neighbor Search With Navigating Spreading-out Graphs. *arXiv preprint arXiv:1707.00143* (2017).

[6] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 541–552.

[7] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 518–529. http://dl.acm.org/citation.cfm?id=645925.671516

[8] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[9] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 604–613.

[10] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 835–850.

[11] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2016. Approximate Nearest Neighbor Search on High Dimensional Data—Experiments, Analyses, and Improvement (v1. 0). *arXiv preprint arXiv:1610.02455* (2016).

[12] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.

[13] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 311–321. http://dl.acm.org/citation.cfm?id=313559.313789