Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures

Wei Dong
wdong@cs.princeton.edu

Moses Charikar
moses@cs.princeton.edu

Kai Li
li@cs.princeton.edu

Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08540, USA

ABSTRACT

K-Nearest Neighbor Graph (K-NNG) construction is an important operation with many web related applications, including collaborative filtering, similarity search, and many others in data mining and machine learning. Existing methods for K-NNG construction either do not scale, or are specific to certain similarity measures. We present *NN-Descent*, a simple yet efficient algorithm for approximate K-NNG construction with arbitrary similarity measures. Our method is based on local search, has minimal space overhead and does not rely on any shared global index. Hence, it is especially suitable for large-scale applications where data structures need to be distributed over the network. We have shown with a variety of datasets and similarity measures that the proposed method typically converges to above 90% recall with each point comparing only to several percent of the whole dataset on average.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]

General Terms

Algorithms, Performance

Keywords

k-nearest neighbor graph, arbitrary similarity measure, iterative method

1. INTRODUCTION

The K-Nearest Neighbor Graph (K-NNG) for a set of objects V is a directed graph with vertex set V and an edge from each $v \in V$ to its K most similar objects in V under a given similarity measure, *e.g.* cosine similarity for text, l_2 distance of color histograms for images, *etc.* K-NNG construction is an important operation with many web related applications: in (user-based) collaborative filtering [1], a K-NNG is constructed by connecting users with similar rating patterns, and used to make recommendations based on the active user's graph neighbors; in content-based search systems, when the dataset is fixed, a K-NNG constructed offline is more desirable than the costly online K-NN search. K-NNG is also a key data structure for many established methods in data mining [6] and machine learning [5], especially manifold learning [23]. Further more, an efficient K-NNG construction method will enable the application of a large pool of existing graph and network analysis methods to datasets without an explicit graph structure.

K-NNG construction by brute-force has cost $O(n^2)$ and is only practical for small datasets. Substantial effort has been devoted in research related to K-NNG construction and K-NN search, and numerous methods have been developed, but existing methods either do not scale, or are specific to certain similarity measures.

Paredes *et al.* [19] proposed two methods for K-NNG construction in general metric spaces with low empirical complexity, but both require a global data structure and are hard to parallelize across machines. Efficient methods for l_2 distance have been developed based on recursive data partitioning [8] and space filling curves [9], but they do not naturally generalize to other distance metrics or general similarity measures.

Indexing data for K-NN search is a closely related open problem that has been extensively studied. A K-NNG can be constructed simply by repetitively invoking K-NN search for each object in the dataset. Various tree-based data structures are designed for both general metric space and Euclidean space [18, 15, 4]. However, they all have the scalability problem mentioned above. Locality Sensitive Hashing (LSH) [13] is a promising method for approximate K-NN search. Such hash functions have been designed for a range of different similarity measures, including hamming distance [13], l_p with $p \in (0, 2]$ [10], cosine similarity [7], *etc.* However, the computational cost remains high for achieving accurate approximation, and designing an effective hash function for a new similarity measure is non-trivial.

In the text retrieval community, methods based on prefix-filtering have been developed for ϵ-NN graph construction, a.k.a. all pairs similarity search or similarity join [2, 22, 21]. An ϵ-NN graph is different from a K-NNG in that undirected edges are established between all pairs of points with a similarity above ϵ. These methods are efficient with a tight similarity threshold, when the ϵ-NN graphs constructed are usually very sparse and disconnected.

Thus, efficient K-NNG construction is still an open problem, and none of the known solutions for this problem is general, efficient and scalable. In this paper, we present *NN-Descent* , a simple but effective K-NNG construction algorithm meeting these requirements with the following properties:

- *General.* Our method works with an arbitrary simi-

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.

larity oracle — a function that produces a similarity score for two objects.

- *Scalable.* As the size of the dataset grows, our method only sees a marginal decline in recall, and the empirical cost is around $O(n^{1.14})$ for all datasets we experimented with. Our method mainly operates on information that is local to each data item, and is intrinsically suitable for a distributed computing environment like MapReduce [11].

- *Space efficient.* In principle, the only data structure we need is an approximate K-NNG which is also the final output: our method can iteratively improve the graph in place. For optimization, or in a distributed implementation, minimal extra data are maintained.

- *Fast and accurate.* We demonstrate with real-life datasets that our method typically converges to above 90% recall with each point comparing only to several percent of the whole dataset on average.

- *Easy to implement.* Our single-node implementation with all optimizations described in this paper takes less than 200 lines of C++ code (excluding I/O and evaluation code).

We compare our method against two existing methods, *i.e.* Recursive Lanczos Partitioning [8] and Locality Sensitive Hashing [13] for the special case of the l_2 metric, and show that our method consistently out-perform those methods.

2. THE PROPOSED METHOD

2.1 Notations and Background

Let V be a dataset of size $N = |V|$, and let $\sigma : V \times V \to \mathbb{R}$ be a similarity measure. For each $v \in V$, let $B_K(v)$ be v's K-NN, *i.e.* the K objects in V (other than v) most similar to v. Let $R_K(v) = \{u \in V \mid v \in B_K(u)\}$ be v's *reverse K-NN*. In the algorithm, we use $B[v]$ and $R[v]$ to store the approximation of $B_K(v)$ and $R_K(v)$, together with the similarity values, and let $\overline{B}[v] = B[v] \cup R[v]$, referred to as the general *neighbors* of v. $B[v]$ is organized as a heap, so updates cost $O(\log K)$.

We are particularly interested in the case when V is a metric space with a distance metric $d : V \times V \to [0, +\infty)$ for which more specific analysis can be done. Since smaller distance means higher similarity, we simply let $\sigma = -d$. For any $r \in [0, +\infty)$, the *r-ball* around $v \in V$ is defined as $B_r(v) = \{u \in V \mid d(u, v) \leq r\}$.

A metric space V is said to be *growth restricted* if there exists a constant c, s.t.

$$|B_{2r}(v)| \leq c|B_r(v)|, \quad \forall v \in V.$$

The smallest such c is called the *growing constant* of V, which is a generalization of the concept of dimensionality and captures the complexity of the dataset.

2.2 The Basic Algorithm

Our method is based on the following simple principle: *a neighbor of a neighbor is also likely to be a neighbor.* In other words, if we have an approximation of the K-NN for each point, then we can improve that approximation by exploring each point's neighbors' neighbors as defined by the current approximation.

This observation can be quantified by the following *heuristic* argument when V is a growth restricted metric space. Let c be the growing constant of V and let $K = c^3$. Assume we already have an approximate K-NNG B, and for every $v \in V$, let $B'[v] = \bigcup_{v' \in B[v]} B[v']$ be the set of points we explore trying to improve B. If the accuracy of B is reasonably good, such that for certain fixed radius r, for all $v \in V$, $B[v]$ contains K neighbors that are uniformly distributed in $B_r(v)$, then *assuming independence of certain events and that* $k \ll |B_{r/2}(v)|$, we can conclude that $B'[v]$ is likely to contain K neighbors in $B_{r/2}(v)$. In other words, we expect to halve the maximal distance to the set of approximate K nearest neighbors by exploring $B'[v]$ for every $v \in V$. This can be seen from the following.

For any $u \in B_{r/2}(v)$ to be found in $B'[v]$, we need to have at least one v' such that $v' \in B[v] \wedge u \in B[v']$. Any $v' \in B_{r/2}(v)$ is likely to satisfy this requirement, as we have:

1. v' is also in $B_r(v)$, so $\Pr\{v' \in B[v]\} \geq K/|B_r(v)|$.
2. $d(u, v') \leq d(u, v) + d(v, v') \leq r$, so $\Pr\{u \in B[v']\} \geq K/|B_r(v')|$.
3. $|B_r(v)| \leq c|B_{r/2}(v)|$, and $|B_r(v')| \leq c|B_{r/2}(v')| \leq c|B_r(v)| \leq c^2|B_{r/2}(v)|$.

Combining 1—3 and assuming independence, we get

$$\Pr\{v' \in B[v] \wedge u \in B[v']\} \geq K/|B_{r/2}(v)|^2$$

In total, we have $|B_{r/2}(v)|$ candidates for such v', so that

$$\Pr\{u \in B'[v]\} \geq 1 - \left(1 - K/|B_{r/2}(v)|^2\right)^{|B_{r/2}(v)|} \approx K/|B_{r/2}(v)|.$$

Let the diameter of the whole dataset be Δ. The above heuristic argument suggests that so long as we pick a large enough K (depending on the growing constant), even if we start from a random K-NNG approximation, we are likely to find for each object K items within a radius $\Delta/2$ by exploring its neighbors' neighbors. The process can be repeated to further shrink the radius until the nearest neighbors are found.

Our basic NN-Descent algorithm, as shown in Algorithm 1, is just a repeated application of this observation. We start by picking a random approximation of K-NN for each object, iteratively improve that approximation by comparing each object against its current neighbors' neighbors, including both K-NN and reverse K-NN, and stop when no improvement can be made.

The approximate K-NNG can be viewed as $K \times N$ functions, each being the distance between one of the N objects and its k-th NN. The algorithm is simply to simultaneously minimize these $K \times N$ functions with the gradient descent method, hence the name "NN-descent". But unlike regular gradient descent, which is applied to a function on $\mathbb{R}^D$ and always explores a small neighborhood of fixed radius, our functions are defined on the discrete set V, and the radius we explore around an object is determined by the previous iteration's approximation of the K-NNG. In fact, the radius starts from a large value as the initial approximation is randomly formed, and shrinks when the approximation is improved through iterations (the number of objects we examine within the radius remains roughly the same). The idea of gradually shrinking search radius through iterations

```
Algorithm 1: NNDescent
  Data: dataset $V$, similarity oracle $\sigma$, $K$
  Result: K-NN list $B$
  begin
      $B[v] \longleftarrow \textsc{Sample}(V, K) \times \{\infty\}, \quad \forall v \in V$
      loop
          $R \longleftarrow \textsc{Reverse}(B)$
          $\overline{B}[v] \longleftarrow B[v] \cup R[v], \quad \forall v \in V;$
          $c \longleftarrow 0$    //update counter
          for $v \in V$ do
              for $u_1 \in \overline{B}[v], u_2 \in \overline{B}[u_1]$ do
                  $l \longleftarrow \sigma(v, u_2)$
                  $c \longleftarrow c + \textsc{UpdateNN}(B[v], \langle u_2, l \rangle)$
          return $B$ if $c = 0$

  function $\textsc{Sample}(S, n)$
  return Sample $n$ items from set $S$

  function $\textsc{Reverse}(B)$
  begin
      $R[v] \longleftarrow \{u \mid \langle v, \cdots \rangle \in B[u]\} \quad \forall v \in V$
      return $R$

  function $\textsc{UpdateNN}(H, \langle u, l, \ldots \rangle)$
  Update K-NN heap $H$; return 1 if changed, or 0 if not.
```

is similar to decentralized search of small-world networks [14] (global optimization). The effect is that most points can reach their true K-NN in a few iterations.

The basic algorithm already performs remarkably well on many datasets. In practice, it can be improved in multiple ways as discussed in the rest of this section.

2.3 Local Join

Given point v and its neighbors $\overline{B}[v]$, a *local join* on $\overline{B}[v]$ is to compute the similarity between each pair of different $p, q \in \overline{B}[v]$, and to update $B[p]$ and $B[q]$ with the similarity. The operation of having each point explore its neighbors' neighbors can be equally realized by a local join on each point's neighborhood, *i.e.* each point introducing its neighbors to know each other. To see that, consider the following relationship: $a \rightarrow b \rightarrow c$, meaning that $b \in B_K(a)$ and $c \in B_K(b)$ (the directions does not matter as we also consider reverse K-NN). In the basic algorithm, we compare a and c twice, once when exploring around either a or c (the redundancy can be avoided by comparing only when $a > c$). Equally, the comparison between a and c is guaranteed by the local join on $\overline{B}[b]$.

Even though the amount of computation remains the same, local join dramatically improves data locality of the algorithm and makes its execution much more efficient. Assume that the average size of $\overline{B}[\cdot]$ is $\overline{K}$, in the basic algorithm, exploring each point's neighborhood touches $\overline{K}^2$ points; the local join on each point, on the contrary, touches only $\overline{K}$ points.

For a single machine implementation, the local join optimization may speed up the algorithm by several percent to several times by improving cache hit rate. For a MapReduce implementation, local join largely reduce data replication among machines.

2.4 Incremental Search

As the algorithm runs, fewer and fewer new items make their way into the K-NNG in each iteration. Hence it is wasteful to conduct a full local join in each iteration as many pairs are already compared in previous iterations. We use the following incremental search strategy to avoid redundant computation:

- We attach a boolean flag to each object in the K-NN lists. The flag is initially marked true when an object is inserted into the list.

- In local join, two objects are compared only if at least one of them is new. After an object participates in a local join, its flag is marked false.

2.5 Sampling

So far there are still two problems with our method. One is that the cost of local join could be high when K is large. Even if only objects in K-NN are used for a local join, cost of each iteration is K^2N similarity comparisons. The situation is worse when reverse K-NN is considered, as there is no limit on the size of reverse K-NN. Another problem is that it is possible that two points are both connected to more than one point, and are compared multiple times when local join is conducted on their common neighbors. We use the sampling strategy to alleviate both problems:

- Before local join, we sample ρK out of the K-NN items marked true for each object to use in local join, $\rho \in (0, 1]$ being the *sample rate*. Only those sampled objects are marked false after each iteration.

- Reverse K-NN lists are constructed separately with the sampled objects and the objects marked false. Those lists are sampled again, so each has at most ρK items.

- Local join is conducted on the sampled objects, and between sampled objects and old items.

Note that the objects marked true but not sampled in the current iteration still have a chance to be sampled in future iterations, if they are not replaced by better approximations.

We found that the algorithm usually converges to acceptable recall even when only a few items are sampled. Both accuracy and cost decline with sample rate ρ, though cost declines much faster (evaluated in Section 4.4.2). The parameter ρ is used to control the trade-off between accuracy and speed.

2.6 Early Termination

The natural termination criterion is when the K-NNG can no longer be improved. In practice, the number of K-NNG updates in each iteration shrinks rapidly. Little real work is done in the final few iterations, when the bookkeeping overhead dominates the computational cost. We use the following early termination criteria to stop the algorithm when further iteration can no longer bring meaningful improvement to accuracy: we count the number of K-NN list updates in each iteration, and stop when it becomes less than δKN, where δ is a precision parameter, which is roughly the fraction of true K-NN that are allowed to be missed due to early termination. We use a default δ of 0.001.

Algorithm 2: NNDescentFull

```
Data: dataset $V$, similarity oracle $\sigma$, $K$, $\rho$, $\delta$
Result: K-NN list $B$
begin
    $B[v] \longleftarrow \textsc{Sample}(V, K) \times \{\langle \infty, true \rangle\} \quad \forall v \in V$
    loop
        parallel for $v \in V$ do
            $old[v] \longleftarrow$ all items in $B[v]$ with a false flag
            $new[v] \longleftarrow \rho K$ items in $B[v]$ with a true flag
            Mark sampled items in $B[v]$ as false;
1       $old' \leftarrow \textsc{Reverse}(old)$, $new' \leftarrow \textsc{Reverse}(new)$
        $c \longleftarrow 0$    //update counter
        parallel for $v \in V$ do
            $old[v] \longleftarrow old[v] \cup \textsc{Sample}(old'[v], \rho K)$
            $new[v] \longleftarrow new[v] \cup \textsc{Sample}(new'[v], \rho K)$
            for $u_1, u_2 \in new[v], u_1 < u_2$
            or $u_1 \in new[v], u_2 \in old[v]$ do
                $l \longleftarrow \sigma(u_1, u_2)$
                // $c$ and $B[.]$ are synchronized.
2               $c \longleftarrow c + \textsc{UpdateNN}(B[u_1], \langle u_2, l, true \rangle)$
3               $c \longleftarrow c + \textsc{UpdateNN}(B[u_2], \langle u_1, l, true \rangle)$
        return $B$ if $c < \delta N K$
```

2.7 The Full Algorithm

The full NN-Descent algorithm incorporating the four optimizations discussed above is listed in Algorithm 2. In this paper we are mainly interested in a method that is independent of similarity measures. Optimizations specialized to particular similarity measures are possible. For example, if the similarity measure is a distance metric, triangle inequality could be potentially used to avoid unnecessary computation.

Our optimizations are not sufficient to ensure that the similarity between two objects is only evaluated once. Full elimination of redundant computation would require a table of $O(N^2)$ space, which is too expensive for large datasets. Space efficient approximations, like Bloom filter, are possible, but come with extra computational cost, and would only be helpful if similarity measure is very expensive to compute.

2.8 On MapReduce Implementation

Our algorithm can be easily implemented under the MapReduce framework. A record consists of a key object and a list of (candidate) neighbors each attached with its distances to the key object. An iteration is realized with two MapReduce operations: first, the mapper issues the input record and the reversed K-NN items, and the reducer merges the K-NN and reverse K-NN; second, the mapper conducts a local join and issues the input record as well as compared pairs of objects, and the reducer merges the neighbors of each key object, keeping only the top K items.

3. EXPERIMENTAL SETUP

This section provides details about experimental setup, including datasets and similarity measures, performance measures, default parameters and system environments. Experimental results are to be reported in the next section.

3.1 Datasets and Similarity Measures

We use 5 datasets and 5 similarity measures, divided into three categories as described below. These datasets are chosen to reflect a variety of real-life use cases, and to cover similarity measures of different natures. Table 1 summarizes the salient information of these datasets.

Four of the datasets are each experimented with two similarity measures (l_1 and l_2, or Jaccard and cosine). Our results show that the two similarity measures for the same dataset usually produce very similar performance numbers and overlapping curves, so in some plots and tables we only report results with one similarity measure per dataset due to space limitation. However, this is not to suggest that different similarity measures for the same dataset are interchangeable. For example, if we test our method with l_1 distance on a benchmark generated with l_2 distance (or vise-versa), only around 70% recall can be reached as apposed to above 95% when the right measure is used.

Dataset	# Objects	Dimension	Similarity Measures
Corel	662,317	14	l_1, l_2
Audio	54,387	192	l_1, l_2
Shape	28,775	544	l_1, l_2
DBLP	857,820	N/A	Cosine, Jaccard
Flickr	100,000	N/A	EMD

Table 1: Dataset summary

3.1.1 *Dense Vectors*

Datasets in this category are composed of dense feature vectors with l_1 and l_2 metrics. These datasets are used in a previous work [12] to evaluate LSH, and the reader is referred to that study for detailed information on dataset construction.

Corel: This dataset contains features extracted from 66,000 Corel stock images. Each image is segmented into about 10 regions, and a feature is extracted from each region. We treat region features as individual objects.

Audio: This dataset contains features extracted from the DARPA TIMIT collection, which contains recordings of 6,300 English sentences. We break each sentence into smaller segments and extract features. Again, we treat segment features as individual objects.

Shape: This dataset contains about 29,000 3D shape models from various sources. A feature is extracted from each model.

3.1.2 *Text Data*

We tokenize and stem text data and view them as, depending on the context, multisets of words, or sparse vectors. We apply two popular similarity measures on text data:

- Cosine similarity (vector view) : $C(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|}$;
- Jaccard similarity (multiset view): $J(x, y) = \frac{|x \cap y|}{|x \cup y|}$.

DBLP: This dataset contains 0.9 million bibliography records from the DBLP web site. Each record includes the authors' full names and the title of a publication. The same dataset was used in a previous study [22] on similarity join of text data.

3.1.3 Earth Mover's Distance

The Earth Mover's Distance (EMD) [1] [20] is a measure of distance between two weighted sets of feature vectors and has been shown effective for content-based image retrieval [16]. Let $P = \{\langle w_i, v_i \rangle\}$ and $Q = \{\langle w_j, v_j \rangle\}$ be two sets of features with normalized weights $\sum_i w_i = \sum_j w_j = 1$, and let d be the feature distance, we use the following definition of EMD:

$$\mathrm{EMD}(P, Q) = \min \sum_i \sum_j f_{ij} d(v_i, v_j)$$

$$s.t. \quad \sum_j f_{ij} = w_i, \quad \sum_i f_{ij} = w_j.$$

Evaluating EMD is costly as it involves solving a linear programming. We use EMD to show the generality of our method.

Flickr: We apply the same feature extraction method [16] as used for the Corel dataset to 100,000 randomly crawled Flickr images. We set the weights of the feature vectors to be proportional to the number of pixels in their corresponding image regions. Following the original paper [16], we use l_1 as feature distance.

3.2 Performance Measures

We use *recall* as an accuracy measure. The ground truth is true K-NN obtained by scanning the datasets in brute force. The recall of one object is the number of its true K-NN members found divided by K. The recall of an approximate K-NNG is the average recall of all objects.

We use the number of similarity evaluations conducted as an architecture independent measure of computational cost. The absolute number depends on the size of the dataset and is not directly comparable across datasets. So we use a normalized version of the measure:

$$\text{scan rate} = \frac{\#\text{ similarity evaluations}}{N(N-1)/2}.$$

The denominator is the total number of possible similarity evaluations of a dataset, and should not be exceeded by any reasonable algorithm.

When scan rate is not an appropriate cost measure for an existing method, we simply compare costs by measuring CPU time.

3.3 Default Parameters

Our method involves three parameters: K, sample rate ρ, and termination threshold δ (note that K would have to be enlarged if the value required by the application is not big enough to produce high recall). Unless otherwise mentioned, we use a default $K = 20$ for all datasets, except that for DBLP we use $K = 50$. The DBLP dataset is intrinsically more difficult than the others and need a larger K to reach about 90% recall. We use a default sample rate of $\rho = 1.0$, *i.e.* we do not conduct sampling except for trimming reverse K-NN to no more than K elements. For some experiments, we also report the performance numbers for $\rho = 0.5$, as a faster but less accurate setting. We use a default termination threshold of 0.001.

[1]Code obtained from `http://www.cs.duke.edu/~tomasi/software/emd.htm`, minor changes made to support parallelization.

Dataset	**Measure**	Default		Fast ($\rho = 0.5$)	
		Recall	**Cost**	**Recall**	**Cost**
Corel	l_1	0.989	0.00817	0.983	0.00467
Corel	l_2	0.997	0.00782	0.995	0.00436
Audio	l_1	0.972	0.0764	0.945	0.0450
Audio	l_2	0.985	0.0758	0.969	0.0445
Shape	l_1	0.995	0.137	0.989	0.0761
Shape	l_2	0.997	0.136	0.994	0.0754
DBLP	Cosine	0.894	0.0271	0.839	0.0146
DBLP	Jaccard	0.886	0.0309	0.848	0.0173
Flickr	EMD	0.925	0.047	0.877	0.0278

Table 2: Overall performance under default setting ($\rho = 1.0$) and a "fast" setting ($\rho = 0.5$), with cost measured in scan rate. The default setting ensures high recall (near or above 90%). When minor loss in recall is acceptable, the fast setting reduces scan rate by nearly half. Shape has a particularly high scan rate due to its small size.

3.4 System Environment

We used commodity servers of the following configuration: dual quad core Intel E5430 2.66GHz CPU; 16GB main memory. All machines ran CentOS 5.3 with Linux kernel 2.6.18 and gcc 4.3.4. We use OpenMP based parallelization for our own code and LSHKIT [2]. The Recursive Lanczos Bisection code [3] is not parallelized and we disabled the parallelization of our code when comparing against it.

4. EXPERIMENTAL RESULTS

This section reports experimental results. We are interested in answering the following questions:

- How does our method perform under typical settings?
- How does our method compare against existing approaches?
- How do accuracy and cost change as dataset scales?
- How to pick a suitable set of parameters?
- How does intrinsic dimensionality affect performance?

The last question is answered by an empirical study with synthetic data.

4.1 Overall Performance

Table 2 summarizes the performance of our method on all the datasets and similarity measures under two typical settings: the default setting ($\rho = 1.0$) achieving highest possible accuracy and a "fast" setting ($\rho = 0.5$) with slightly lower accuracy. We see that even with the fast setting, our method is able to achieve $\geq$ 95% recall, except for DBLP and Flickr. for which recall is below 90%. By putting in more computation with the default setting, we are able to boost recall for the more difficult datasets to close or above 90%.

We see that scan rate has a larger variation across datasets, ranging from below 0.01 for Corel to 0.15 for Shape. Multiple factors could affect scan rate, but we will show (Section 4.3) that the size of the dataset is the dominant factor,

[2]Code available at `http://lshkit.sourceforge.net/`.
[3]Code obtained from `http://www.mcs.anl.gov/~jiechen/research/software.html`.

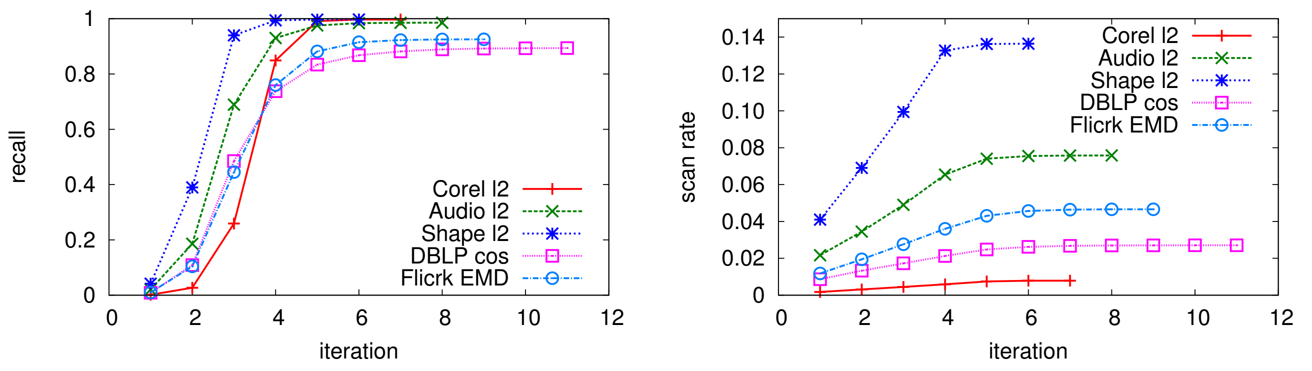


Figure 1: Recall and scan rate vs. iterations. Recall increases fast in the beginning and then quickly converges.

and with all other parameters fixed, scan rate shrinks as dataset grows. The scan rate of Shape is relatively high mainly because its size is small. In general, at million-object level, we expect to cover several percents of the total $N(N-1)/2$ comparisons.

For a closer observation of the algorithm's behavior, Figure 1 shows the accumulative recall and scan rate through iterations. The curves of different data have very similar trends. We see a fast convergence speed across all datasets — the curves are already close to their final recall after five iterations, and all curves converge within 12 iterations.

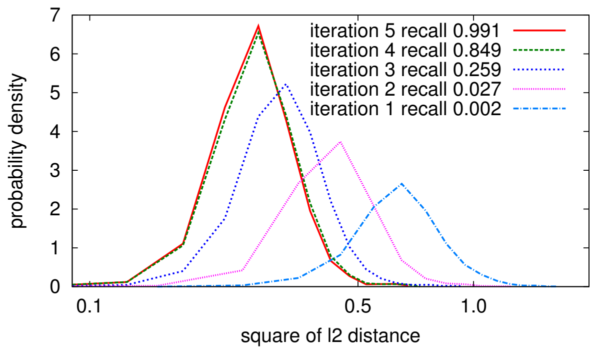


Figure 2: Approximate K-NN distance distributions of Corel dataset after each iteration. The peaks of the bottom three curves spread equally on the log-scaled horizontal axis, suggesting the exponential reduction of the radius covered by approximate K-NN during the execution of our method.

Figure 2 shows the approximate K-NN distance ($N \times K$ distance values) distributions of the Corel datasets during the first five iterations. The peaks of the first three iterations spread equally on a log-scaled horizontal axis, *i.e.* the search radius around each object shrink exponentially in the initial iterations. This remotely confirms our observation made in Section 2.

4.2 Comparison Against Existing Methods

We compare our method with two recent techniques, both specific to l_2 distance, so only the three dense-vector datasets are used. The brute-force approach, even though achieving 100% accuracy, is too expensive for large datasets and is not considered here.

4.2.1 *Recursive Lanczos Bisection*

Recursive Lanczos Bisection (RLB)[8] is a divide-and-conquer method for approximate K-NN graph construction in Euclidean space. According to the two configurations supported by RLB, we conduct comparison under two settings, one for speed ($R = 0.15$ for RLB and $\rho = 0.5$ for ours) and one for quality ($R = 0.3$ for RLB and $\rho = 1.0$ for ours). Table 3 summarizes the recall and CPU time of both methods under those settings. Our method consistently achieves both higher recall and faster speed (2× to 16× speedup) in all cases. Actually, even the recall of our low-accuracy setting beats RLB's high-accuracy setting in all cases except for the Corel dataset, where there is only a difference of 0.001.

Dataset	Method	For Speed		For Accuracy	
		recall	time	recall	time
Corel	RLB	0.988	1844s	0.996	5415s
	Ours	0.995	252s	0.997	335s
Audio	RLB	0.906	84.6s	0.965	188.6s
	Ours	0.969	21.1s	0.986	31.5s
Shape	RLB	0.961	29.7s	0.989	56.0s
	Ours	0.994	14.0s	0.997	24.4s

Table 3: Comparison against Recursive Lanczos Bisection (RLB) under two settings. Our method runs 2 to 16 times faster with consistently higher recall.

4.2.2 *Locality Sensitive Hashing*

LSH is a promising method for approximate K-NN search in high dimensional spaces. We use LSH for offline K-NNG construction by building an LSH index (with multiple hash tables) and then running a K-NN query for each object. We use plain LSH [13] rather than the more recent Multi-Probing LSH [17] in this evaluation as the latter is mainly to reduce space cost, but could slightly raise scan rate to achieve the same recall. We make the following optimizations to the original LSH method to better suit the K-NNG construction task:

- For each query, we use a bit vector to record the objects that have been compared, so if the same points are seen in another hash table, they are not evaluated again.

Dataset	LSH		Ours	
	recall	scan rate	recall	scan rate
Corel	0.906	0.004	0.995	0.004
Audio	0.615	0.047	0.969	0.045
Shape	0.925	0.076	0.994	0.075

Table 4: Comparison against LSH. We achieve much higher recall at similar scan rate. It is impractical to tune LSH to reach our recall as it would become slower than brute-force search.

- We simultaneously keep the approximate K-NN lists for all objects, so whenever two objects are compared, the K-NN lists of both are updated;
- A query is only compared against objects with a smaller ID.

These optimizations eliminate all redundant computations without affecting recall. We translate the cost of LSH into our scan rate measure, so the two methods are directly comparable (we ignore the cost of LSH index construction though).

It is hard for LSH to achieve even the recall of our low-accuracy settings, as the cost would be higher than brute-force search. Hence we tune the scan rate of both methods to an equal level and compare recall. For LSH, we use the same default parameters in our previous study [12], except that we tune the number of hash tables to adjust scan rate. For our method, the low-accuracy setting is used ($\rho = 0.5$).

Table 4 summarizes recall and scan rate for both method. We see that our method strictly out-performs LSH: we achieve significantly higher recall at similar scan rate. Also note that the space cost of LSH is much higher than ours as tens of hash tables are needed, and the computational cost to construct those hash tables are not considered in the comparison.

4.3 Performance as Data Scales

It is important that an algorithm has a consistent accuracy and a predictable cost as data scales, so that parameters tuned with a small sample are applicable to the full dataset. To study the scalability of our algorithm, we run experiments on subsamples of the full datasets with fixed parameter settings and observe the changes in recall and scan rate as sample size grows.

Table 5 shows recall vs. sample size. We see that as dataset grows, there is only a minor decline in recall. This confirms the feasibility of parameter tuning with a sampled dataset.

Figure 3 plots scan rate vs. dataset size, in log-log scale, and we see a very interesting phenomenon: all curves form parallel straight lines, and the curves of all datasets except DBLP almost coincide. This suggests that our method has a polynomial time complexity disregard the complexity of the dataset (which affects the accuracy rather than speed when parameters are fixed). Table 6 shows the empirical complexity of our method on various datasets obtained by fitting the scan rate curves, which is roughly $O(n^{1.14})$ for all datasets.

The main reason that the DBLP curve is higher is that we use $K = 50$ for DBLP and $K = 20$ for other datasets. As a local join costs $O(K^2)$, we expect the scan rate of DBLP to be about $(50/20)^2 = 6.25$ times the scan rate of other datasets if they all converge at the same speed. The real value we estimate from the curves (by dividing the intercept of the DBLP curve and the average intercept of the rest) is 5.2, which is close to the expected value.

Size	Corel	Audio	Shape	DBLP	Flickr
	l_2	l_2	l_2	cos	EMD
1K	1.000	0.999	1.000	0.959	0.999
5K	1.000	0.996	0.992	0.970	0.991
10K	1.000	0.993	0.998	0.970	0.983
50K	0.999	0.988	-	0.951	0.953
100K	0.999	-	-	0.940	0.925
500K	0.997	-	-	0.907	-

Table 5: Recall vs. dataset size. The impact of data size growth is small.

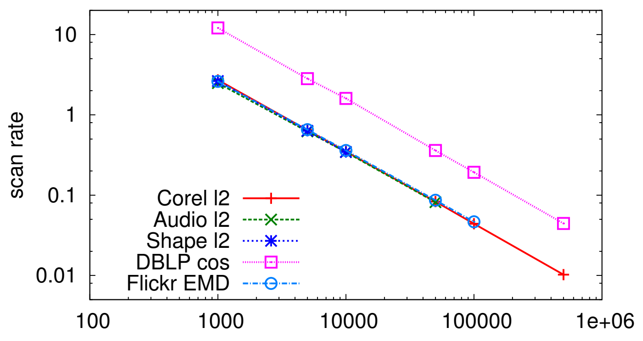


Figure 3: Scan rate vs. dataset size. The conincidence of various datasets (except DBLP) suggests that when parameters are fixed, the time complexity of our method depends polynomially on the dataset size, but is independent on the complexity of the dataset (DBLP is different due to a larger K value used).

4.4 Parameter Tuning

We need to fix three parameters for the algorithm: K, sample rate ρ and termination threshold δ. The meaning of δ is clear: the loss in recall tolerable with early termination. Here we study the impact of K and ρ on performance.

4.4.1 *Tuning K as a Parameter*

The application determines the minimal K required. Meanwhile, a sufficiently large K is necessary to achieve high re-

Dataset & Measure	Empirical Complexity
Corel/l_2	$O(n^{1.11})$
Audio/l_2	$O(n^{1.14})$
Shape/l_2	$O(n^{1.11})$
DBLP/cos	$O(n^{1.11})$
Flickr/EMD	$O(n^{1.14})$

Table 6: Empirical complexity of different datasets and similarity measures under default parameter settings. The values are obtained by fitting the scan rate curves in Figure 3.

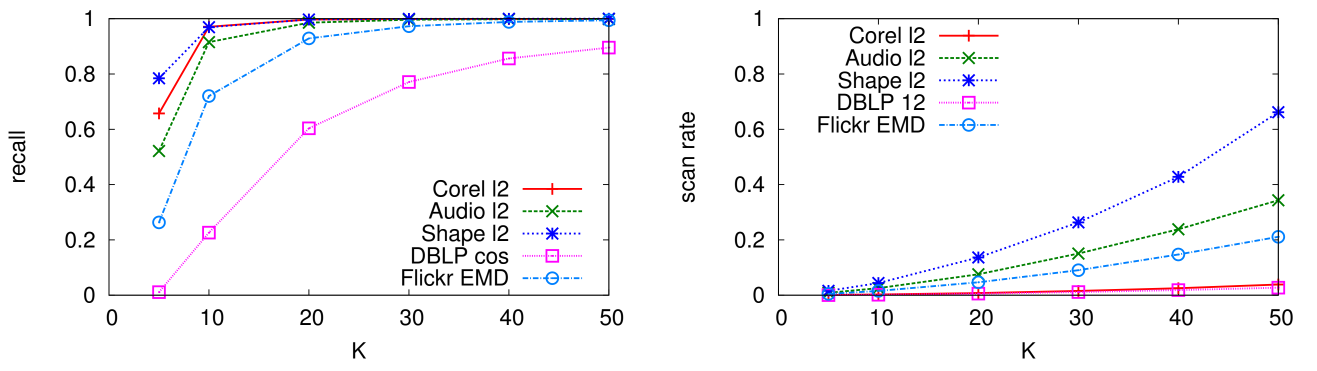


Figure 4: Recall and scan rate vs. K. For a particular dataset, a sufficiently large K is needed for >90% recall. Beyond that, recall only improves marginally.

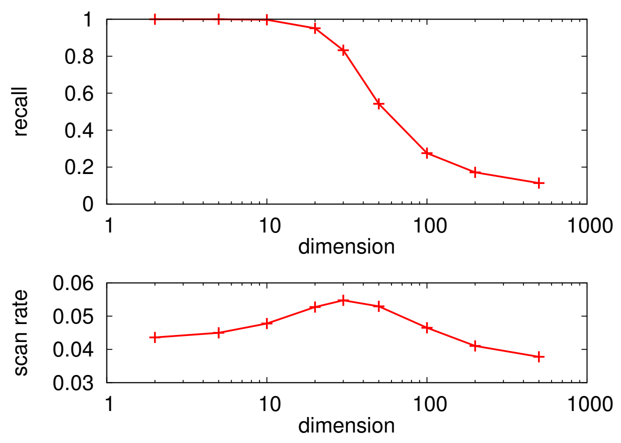


Figure 6: Recall and scan rate vs. dimensionality with $K = 20$. There is a limit of dimensionality up to which the algorithm is able to achieve high recall.

call. The minimal value required by the application might need to be enlarged.

Figure 4 plots the relationship between K and performance. We see that $K \geq 10$ is needed for the dense vector datasets to achieve good recall, and for text, recall reaches 90% only with $K \geq 50$. This suggests that text has a high intrinsic dimensionality than the other datasets. We also see that beyond a critical point, recall only improves marginally as K further grows.

4.4.2 *Sample Rate for Accuracy-Cost Trade-Off*

The sample rate ρ can be used to control accracy-cost trade-off. Figure 5 plots sample rate vs. performance. We see that even with a sample rate of 0.1, reasonably high recall can be achieved for most datasets, and recall grows very slowly beyond $\rho = 0.5$. Scan rate, on the other hand, has a near-linear relationship with ρ across the whole range. We suggest $\rho = 0.5$ for applications without a critical dependent on high recall.

4.5 Impact of Intrinsic Dimensionality

We use synthetic data to study the impact of dimensionality on the performance of our algorithm, as the intrinsic dimensionality of real-life datasets is not obvious and can not be controlled. We generate a D dimensional vector by concatenating D independent random values sampled from the uniform distribution $U[0, 1]$. Data generated in this way have the same intrinsic dimensionality and full dimensionality. We then test the performance of our method with different dimensionality, each with a dataset size of $100,000$.

Figure 6 plots the performance of our algorithm, with default setting, under different dimensionality with a fixed $K = 20$. We see that recall decreases as dimensionality increases, and our method performs well (recall $> 95\%$) with dimensionality ≤ 20 (which happens to be the value of K). Beyond that, recall rapidly drops to about 50% as dimensionality grows to 50, and eventually approaches 0 as dimensionality further grows.

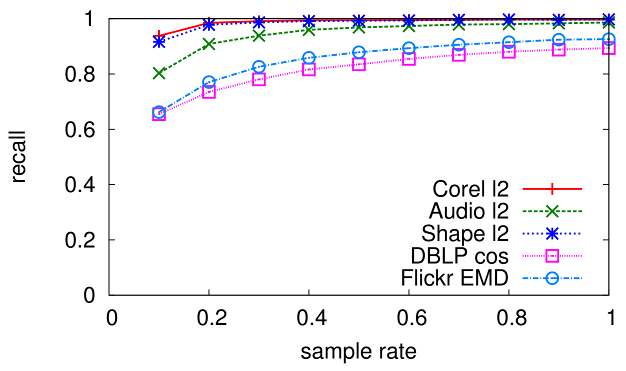
The impact of dimensionality on cost, although not as large, is also interesting. When dimension is low and most points are able to reach their true K-NN (global optima), cost is low. Convergence speed slows as dimensionality increases and scan rate also increases. Scan rate peaks at around 30 dimensions. After that, most points are not able to reach their true K-NN and get more and more easily trapped at local optima, so scan rate begins to shrink as dimensionality further grows. Overall, the fluctuation of cost is low, within a range of $2\times$.

We then study how recall can be improved by enlarging K. Table 7 summarizes the recall and scan rate of representative K values at various dimensionality. The results can be categorized into three zones of dimensionality:

- Small dimensionality ($D = 2, 5$): extremely high recall (close to 1) and very low scan rate (< 0.01) can be achieved.

- Medium dimensionality ($D = 10, 20$): recall reaches 95% with $K = D$; scan rate is relatively higher, around 5% (due to a larger K). Recall still increases as K grows, but a recall close to 1 is no longer practical as scan rate would grow too high.

- Large dimensionality ($D = 50, 100$): recall peaks at $K = 50$ and declines beyond that, and the peak recall shrinks as D grows (94% for $D = 50$ and 78% for $D = 100$). Scan rate is around 1/4, already too high for the algorithm to be practically useful.



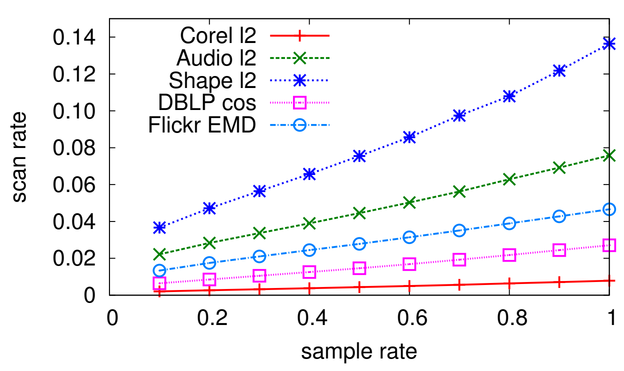


Figure 5: Recall and scan rate vs. sample rate ρ. Recall grows marginally after $\rho > 0.5$ while cost grows in a near-linear fashion across the whole range of ρ.

D = 2			D = 5			D = 10			D = 20			D = 50			D = 100		
K	recall	cost	K	recall	cost	K	recall	cost	K	recall	cost	K	recall	cost	K	recall	cost
4	0.875	0.004	5	0.862	0.006	9	0.925	0.014	19	0.942	0.0487	49	0.934	0.238	49	0.774	0.240
5	0.990	0.005	6	0.957	0.007	10	0.950	0.016	20	0.952	0.0527	50	0.939	0.245	50	0.781	0.248
6	0.996	0.006	7	0.980	0.009	11	0.967	0.019	21	0.957	0.0569	51	0.925	0.253	51	0.778	0.257

Table 7: Tuning K for various dimensionality. For a dataset of fixed size and when dimensionality is high ($D \geq 50$), recall cannot always be improved by enlarging K, and the highest achievable recall shrinks as dimensionality grows.

This suggest that our method is best applied to dataset with intrinsic dimensionality around 20. It still works, but with relatively high cost, for dimensionality around 50, and start to fail as dimensionality further grows. Fortunately, all datasets we experimented with, and actually most real-life datasets where K-NN is meaningful [3], are of relatively low dimensionality.

5. RELATED WORKS

Paredes *et al.* [19] are the first to study K-NNG construction for general metric space as a primary problem (rather than an application of K-NN search methods). Some general observations they made also apply to our work: the K-NNG under construction can be used to improve the remaining construction work and cost can be reduced by solving the N K-NN queries jointly. They solved the K-NNG construction problem in two stages: first an index is built, either a tree structure or a pivot table, and then the index is used to solve a K-NN query for each object. Despite the high level similarity to using a general K-NN search index for K-NNG construction, strategies to exploit the approximate K-NNG already constructed are incorporated to the search process. They also studied the empirical complexity of their methods. For example, the pivot based method achieves a better empirical complexity, which is $O(n^{1.10})$ at 4 dimensions and $O(n^{1.96})$ at 24 dimensions.

Efficient K-NNG construction methods have been developed specifically for l_2 metric. Recursive Lanczos Bisection [8] uses inexpensive Lanczos procedure to recursively divide the dataset, so objects in different partitions do not have to be compared. Connor *et al.* [9] used space filling curve to limit the search range around each object, and an extra verification and correction stage is used to ensure accuracy. These methods do not easily generalize to other distance metrics or general similarity measures.

The K-NN search problem is closely related to K-NNG construction. After all, if the K-NN search problem is solved, K-NNG can be constructed simply by running a K-NN query for each object. For datasets of small dimensionality, various tree data structures [18, 15, 4] can be used to efficiently solve the problem. K-NN search in high dimensional spaces is still and open problem, and the most promising approach is to solve the problem approximately with Locality Sensitive Hashing [13, 17]. As we have shown with experiments, it is hard for LSH to achieve high recall, and designing an affective hash function for a new similarity measure is non-trivial.

In the text retrieval community, efficient methods based on prefix-filtering are developed for the ϵ-NN graph construction [2, 22, 21], a different kind of nearest neighbor graph which establishes an edge between all pairs of points whose similarity is above ϵ. The problem is that such methods are only efficient for a very tight similarity threshold, corresponding to a very sparse and disconnected graph.

6. CONCLUSION

We presented *NN-Descent* , a simple and efficient method for approximate K-Nearest Neighbor graph construction with arbitrary similarity measures, and demonstrated its excellent accuracy and speed with extensive experimental study. Our method has a low empirical complexity of $O(n^{1.14})$ (on various tested datasets) and can be easily parallelized, potentially enabling the application of existing graph and network analysis methods to large-scaled dataset without an explicit graph structure. Rigorous theoretical analysis of our method is an interesting problem to be solved.

Acknowledgments

This work is supported in part by Gigascale Systems Research Center, Google Research Grant, Yahool Research Grant, Intel Research Council, by National Science Foundation grants CSR-0509447 and CSR-0509402.

7. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749, 2005.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 131–140, 2007.

[3] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 217–235, 1999.

[4] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 97–104, 2006.

[5] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *CVPR '08: Proceedings of the 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2008.

[6] M. R. Brito, E. L. Chávez, A. J. Quiroz, and J. E. Yukich. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35(1):33–42, August 1997.

[7] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[8] J. Chen, H. ren Fang, and Y. Saad. Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection. *Journal of Machine Learning Research*, 10:1989–2012, 2009.

[9] M. Connor and P. Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16:599–608, 2010.

[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 669–678, 2008.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, 1999.

[14] J. Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170, 2000.

[15] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 17*. 2005.

[16] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 208–217, 2004.

[17] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-Probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 950–961, 2007.

[18] A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *In Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 397–405, 2000.

[19] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of -nearest neighbor graphs in metric spaces. In *WEA*, pages 85–97, 2006.

[20] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 59, 1998.

[21] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 495–506, 2010.

[22] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 131–140, 2008.

[23] S. Yan, D. Xu, B. Zhang, H.-J. Zhang, Q. Yang, and S. Lin. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29:40–51, 2007.