Bi-level Locality Sensitive Hashing for K-Nearest Neighbor Computation

Jia Pan
UNC Chapel Hill
panj@cs.unc.edu

Dinesh Manocha
UNC Chapel Hill
dm@cs.unc.edu

ABSTRACT

We present a new Bi-level LSH algorithm to perform approximate k-nearest neighbor search in high dimensional spaces. Our formulation is based on a two-level scheme. In the first level, we use a RP-tree that divides the dataset into subgroups with bounded aspect ratios and is used to distinguish well-separated clusters. During the second level, we construct one LSH hash table for each sub-group, which is enhanced by a hierarchical structure based on space-filling curves and lattice techniques. Given a query, we first determine the sub-group that it belongs to and then perform a k-nearest neighbor search within the suitable buckets in the LSH hash table corresponding to the sub-group. In practice, our algorithm is able to improve the quality and reduce the runtime of approximate k-nearest neighbor computations. We demonstrate the performance of our method on two large, high-dimensional and widely used image datasets and show that when given the same runtime budget, our bi-level method can provide better accuracy in terms of recall or error ratio as compared to prior methods. Moreover, our approach reduces the variation in runtime cost or the quality of results on different datasets.

1. INTRODUCTION

Nearest neighbor search in high-dimensional space is an important problem in database management, data mining, computer vision and search engine. The underlying applications use feature-rich data, such as digital audio, images or video, which are typically represented as high-dimensional feature vectors. One popular way to perform similarity searches on these datasets is via an exact or approximate k-nearest neighbor search in a high-dimensional feature space. This problem is well studied in literature, and is regarded as a challenging problem due to its intrinsic complexity and the quality or accuracy issues that arise in terms of computing the appropriate k-nearest neighbors.

In terms of runtime cost, ideal nearest neighbor query should take $\mathcal{O}(1)$ or $\mathcal{O}(\lg n)$ per-query time, because the size of the dataset (i.e. n) can be very large (e.g. > 1 million). Moreover, the space required should be $\mathcal{O}(n)$ in order to handle large datasets. In terms of quality issues, each query should return k-nearest neighbor results that are close enough to the exact k-nearest neighbors computed via a brute-force, linear-scan approach that has a high $O(n)$ per-query complexity. In addition, 'concentration effect' tends to appear in high-dimensional datasets, i.e. the distances between the nearest and to the farthest neighbors become indiscernible when data dimensionality increases [3].

Most current approaches for k-nearest neighbor computation are unable to satisfy the runtime requirements for high-dimensional datasets. For example, tree-based methods such as cover-tree [4], SR-tree [17] can compute accurate results, but are not time-efficient for high-dimensional data. When the dimensionality exceeds 10, these space partitioning-based methods can be slower than the brute-force approach [30]. Approximate nearest neighbor algorithms tend to compute neighbors that are close enough to the queries instead of the exact k-nearest neighbors, and have a lower runtime and memory overhead than the exact algorithms [18]. For high-dimensional k-nearest neighbor search, one of the widely used approximate methods is *locality-sensitive hashing* (LSH) [7], which uses a family of hash functions to group or collect nearby items into the same bucket with a high probability. In order to perform a similarity query, LSH-based algorithms hash the query item into one bucket and use the data items within that bucket as potential candidates for the final results. Moreover, the items in the bucket are ranked according to the exact distance to the query item in order to compute the k-nearest neighbors. The final ranking computation among the candidates is called the *short-list search*, which is regarded as the main bottleneck in LSH-based algorithms. In order to achieve high search accuracy, LSH-based methods use multiple (e.g. more than 100 [12]) hash tables, which results in high space and time complexity. To reduce the number of hash tables, some LSH variations tend to use more candidates [22], estimate optimal parameters [2, 9] or use improved hash functions [1, 14, 25, 13]. All these methods only consider the average runtime or quality of the search process over randomly sampled hash functions, which may result in large deviations in the runtime cost or the quality of k-nearest neighbor results.

Main Results: In this paper, we present a novel bi-level scheme based on LSH that offers improved runtime and quality as compared to prior LSH-based algorithms. We use a two-level scheme and during the first level, we use

VLDB '10, September 13-17, 2010, Singapore

random projections [11] as a preprocess to divide the given dataset into multiple subgroups with bounded aspect ratios (i.e. roundness) [8] and can distinguish well-separated clusters. This preprocess step tends to decrease the deviation in the runtime and quality due to the randomness of the LSH framework and the non-uniform distribution of data items. During the second level, we apply standard [7] or multi-probe [22] LSH techniques to each subgroup and compute the approximate k-nearest neighbors. In order to reduce the runtime/quality deviation caused by different queries, we construct a hierarchical LSH table based on Morton curves [5]. The hierarchy is also enhanced by lattice techniques [14] to provide better k-nearest neighbor candidates for LSH short-list search. We show the improvements over prior LSH schemes using well-known metrics that are widely used to evaluate the performance of LSH algorithms, including selectivity, recall ratio and error ratio. We have applied our new algorithm to two large, high-dimensional image datasets and compared the performance with prior methods. In particular, our method has higher recall and error ratios as compared to prior LSH algorithms, for a given runtime budget measured in terms of selectivity. Moreover, we also show that our scheme can reduce the runtime/quality deviation caused by random projections or different queries.

The rest of the paper is organized as follows. We survey the background of LSH and related work in Section 2. Section 3 gives an overview of our approach. We present our new Bi-level LSH algorithm in Section 4 and analyze its properties in Section 5. We highlight the performance on different benchmarks in Section 6.

2. BACKGROUND AND RELATED WORK

In this section, we briefly review related work on LSH based k-nearest neighbor computation.

2.1 Basic LSH

Given a metric space $(\mathbb{X}, \|\cdot\|)$ and a database $S \subseteq \mathbb{X}$, for any given query $\mathbf{v} \in \mathbb{X}$, the k-nearest neighbor algorithm computes a set of k points $I(\mathbf{v}) \subseteq S$ that are closest to $\mathbf{v}$. We assume that $\mathbb{X}$ is embedded in a D-dimensional Euclidean space $\mathbb{R}^D$ and each item is represented as a high-dimensional vector, i.e. $\mathbf{v} = (v_1, ..., v_D)$.

The basic LSH algorithm is an approximate method to compute k-nearest neighbors, which uses M ($M \ll D$) hash functions $h_1(\cdot), ..., h_M(\cdot)$ to transform $\mathbb{R}^D$ into a lattice space $\mathbb{Z}^M$ and distribute each data item into one lattice cell:

$$H(\mathbf{v}) = \langle h_1(\mathbf{v}), h_2(\mathbf{v}), ..., h_M(\mathbf{v}) \rangle. \quad (1)$$

The lattice space is usually implemented as a hash table, since many of the cells may be empty. LSH algorithms have been developed for several distance measures including Hamming distance and l_p distance. For Hamming space [12], $h_i(\cdot)$ is constructed by first transferring $\mathbf{v}$ into a binary representation and then randomly choosing several bits from the binary representation. For l_p space, $p \in (0, 2]$ [7],

$$h_i(\mathbf{v}) = \lfloor \frac{\mathbf{a}_i \cdot \mathbf{v} + b_i}{W} \rfloor, \quad (2)$$

where the D-dimensional vector $\mathbf{a}_i$ consists of i.i.d. entries from Gaussian distribution $N(0, 1)$ and b_i is drawn from uniform distribution $U[0, W)$. M and W control the dimension and size of each lattice cell and therefore control the locality sensitivity of the hash functions. In order to achieve high quality results, L hash tables are used with independent dim-M hash functions $H(\cdot)$. Given one query item $\mathbf{v}$, we first compute its hash code using $H(\mathbf{v})$ and locate the hash bucket that contains it. All the points in the bucket will belong to its potential k-nearest neighbor candidate set and we represent that set as $A(\mathbf{v})$. Next, we perform a local scan on $A(\mathbf{v})$ to compute the k-nearest neighbors $I(\mathbf{v})$.

There are several known metrics used to measure the performance of a k-nearest neighbor search algorithm. First is the *recall ratio*, i.e. the percentage of the actual k-nearest neighbors $N(\mathbf{v})$ in the returned results $I(\mathbf{v})$:

$$\rho(\mathbf{v}) = \frac{|N(\mathbf{v}) \cap I(\mathbf{v})|}{|N(\mathbf{v})|} = \frac{|N(\mathbf{v}) \cap A(\mathbf{v})|}{|N(\mathbf{v})|}, \quad (3)$$

where $N(\mathbf{v})$ can be computed using any exact k-nearest neighbor approach and serves as the ground-truth.

The second metric is the *error ratio* [12], i.e. the relationship between $\mathbf{v}$'s distance to $N(\mathbf{v})$ and $I(\mathbf{v})$:

$$\kappa(\mathbf{v}) = \frac{1}{k} \sum_{i=1}^{k} \frac{\|\mathbf{v} - N(\mathbf{v})_i\|}{\|\mathbf{v} - I(\mathbf{v})_i\|}, \quad (4)$$

where $N(\mathbf{v})_i$ or $I(\mathbf{v})_i$ is $\mathbf{v}$'s i-th nearest neighbor in $N(\mathbf{v})$ or $I(\mathbf{v})$. We use recall and error ratios to measure the quality of LSH algorithm and we prefer k-nearest neighbor results with large recall and error ratios.

The final metric is the *selectivity* [9], which measures the runtime cost of the short-list search:

$$\tau(\mathbf{v}) = |A(\mathbf{v})|/|S|, \quad (5)$$

where $|S|$ is the size of the dataset. Notice that selectivity is closely related to the *precision* $\sigma(\mathbf{v}) = \frac{|A(\mathbf{v}) \cap N(\mathbf{v})|}{|A(\mathbf{v})|}$, which measures how well hash functions can isolate points far away from each other. We have the relationship $\tau(\mathbf{v}) = \frac{\rho(\mathbf{v}) \cdot k}{\sigma(\mathbf{v}) \cdot |S|}$: time complexity is proportional to the recall ratio and inversely proportional to the precision.

The basic LSH scheme has several drawbacks. First, in practice it needs a large number of hash tables (L) to achieve high recall ratio and low error ratio and this results in high selectivity. Secondly, it is difficult to choose parameters (e.g. M, W for l_p) that are suitable for a given dataset and different queries, which may result in large deviation in runtime and quality. Finally, the short-list search within each bucket can be time consuming when we use large selectivity to obtain a high recall ratio. These drawbacks can affect the efficiency and quality of basic LSH-based algorithms on large datasets.

2.2 Variations of LSH

Many techniques have been proposed to overcome some of the drawbacks of the basic LSH algorithm. For Hamming space, LSH-forest [2] avoids tuning of the parameter M by representing the hash table as a prefix tree and the parameter M is computed based on the depth of the corresponding prefix-tree leaf node. Multi-probe LSH [22] systematically probes the buckets near the query points in a query-dependent manner, instead of only probing the bucket that contains the query point. It can obtain higher recall ratio with fewer hash tables, but may result in larger selectivity from additional probes. Dong et al. [9] construct a statistical quality and runtime model with a small sample dataset, and then compute M and W that can result in a good balance between high recall and low selectivity. The

underlying model assumes the dataset is homogeneous and requires high-quality sampling method to accurately capture the data distribution. Joly et al. [15] improve the multi-probe LSH by using prior information collected from sampled dataset. Tao et al. [28] represent hash table as one LSB-tree and compute the bucket probing sequence based on the longest common prefix rule. Kang et al. [16] use hierarchial and non-uniform bucket partitioning to handle the non-homogeneous datasets. Morever, most LSH approaches focus on average runtime and quality, and do not consider the variations of runtime or quality caused by random selected hash functions.

Many approaches have been proposed to design better hash functions. $\mathbb{Z}^M$ lattice may suffer from the curse of dimensionality: in a high dimensional space, the *density* of $\mathbb{Z}^M$ lattice, i.e. the ratio between the volume of one $\mathbb{Z}^M$ cell and the volume of its inscribed sphere, increases very quickly when the dimensionality increases. In order to overcome these problems, lattices with density close to one are used as space quantizers, e.g. E_8-lattice [14] and Leech lattice [1] are used for dim-8 and dim-24 data items, respectively. Paulevé et al. [24] present a hashing scheme based on K-means clustering which makes the hashing process adaptive to datasets with different distributions.

2.3 Random Projection

LSH scheme is based on the theory of random projections, which can map a set of points in a high-dimensional space into a lower-dimensional space in such a way that distances between the points are almost preserved, as governed by Johnson-Lindenstrauss theorem [29]. As a result, random projections have been widely used in space embedding or dimension reduction, which can be viewed as variations of LSH with different hashing function forms.

Recently, many algorithms have been proposed for Hamming space embedding. Spectral hashing [31] generates hashing rules using spectral graph partitioning. Raginsky and Lazebnik [25] use a hash function based on random Fourier features [26] and is similar to [31].

Some other methods try to design compact Hamming code that is optimized for given database using learning algorithms. Kernel embedding [19] [20] methods tend to learn optimal hashing functions using supervised distances on a small subset of the datasets. Semantic hashing [27] uses stacked restricted Boltzmann machine to learn hash functions that minimize the 'energy' on a dataset. He et al. [13] improve spectral hashing by representing hashing function explicitly using kernel representations.

3. OVERVIEW

In this section, we give an overview of our new Bi-level LSH scheme.

The overall pipeline of our algorithm is shown in Figure 1 and includes two levels. In the first level, we construct a *random projection tree* (RP-tree) [11, 6], which is a space-partitioning data structure that is used to organize the points into several subsets. In a RP-tree, each subset is represented as one leaf node. As compared to other methods such as a Kd-tree or K-means algorithm, RP-tree has many good properties [6, 8]. These include fast convergence speed, guaranteed 'roundness' of leaf nodes, etc. These properties are useful for generating compact LSH hash code and reduce the algorithm's performance/quality variance.

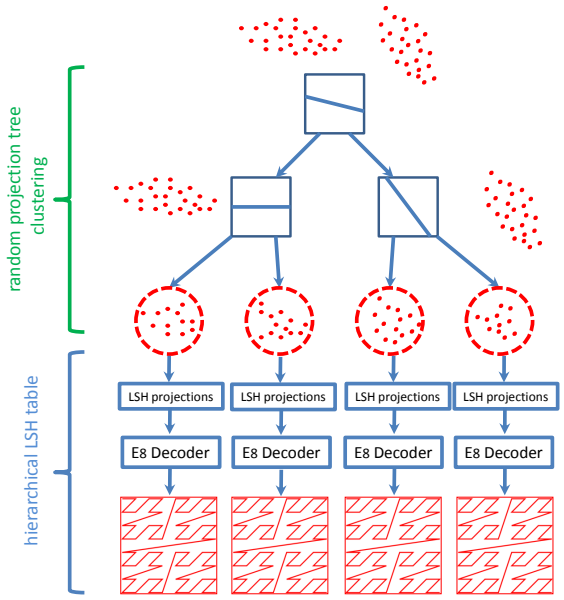


Figure 1: The framework for Bi-level LSH. The first level is the random projection tree. The second level is the hierarchical lattice for each RP-Tree leaf node.

During the second level, we construct a locality-sensitive hash (LSH) table for each of the subsets generated during the first level. Unlike prior methods, our LSH table has a hierarchical structure that is constructed using a Morton curve. The hierarchal LSH table can reduce the runtime performance and quality variance among different queries: for query within regions with high data density, the algorithm only needs to search nearby buckets; for query within regions with low data density, the algorithm can automatically search in far away buckets to provide sufficient number of k-nearest neighbor candidates so that the obtained k-nearest neighbor results will have enough quality. We also enhance the hierarchy using an E_8 lattice, which can overcome $\mathbb{Z}^M$ lattice's drawbacks for high-dimensional datasets and can improve the quality of the results.

For each item $\mathbf{v}$ in the dataset, our method computes the RP-tree leaf node RP-tree($\mathbf{v}$) that contains $\mathbf{v}$ and computes its LSH code $H(\mathbf{v})$ using LSH parameters for that subset in the second level. As a result, our Bi-level LSH scheme decomposes the basic LSH code into two parts: the RP-tree leaf node index and the LSH code corresponding to the subset, i.e. $\tilde{H}(\mathbf{v}) = (\text{RP-tree}(\mathbf{v}), H(\mathbf{v}))$. Later we show in Section 5, that our approach can provide a more compact hash code as compared to the basic LSH algorithm. $\tilde{H}(\mathbf{v})$ is stored in a hash table.

Given one query item, we first traverse the RP-tree to find the leaf node the query belonging to, and then compute its LSH code within that subset. Based on the decomposed hash code, we compute the buckets in the hash table with same or similar hash codes and the elements in these buckets are used as candidates for k-nearest neighbors.

4. BI-LEVEL LOCALITY-SENSITIVE HASHING

In this section, we present the details of our Bi-level LSH framework.

4.1 Level I: RP-tree

During the first level, we use *random projection tree* (RP-tree) [11, 6] to divide the dataset into several small clusters with good properties for subsequent LSH-based operations. We first give a brief overview of RP-trees and present the details of our first-level algorithm. Finally, we analyze the advantages of our first-level preprocessing scheme within the overall Bi-level LSH algorithm.

4.1.1 Background

The RP-tree construction algorithm is similar to Kd-tree computation: given a data set, we use a split rule to divide it into two subsets, which are split recursively until the resulting tree has a desired depth. Kd-tree chooses a coordinate direction (typically the coordinate with the largest spread) and then splits the data according to the median value for that coordinate. Unlike Kd-tree, RP-tree projects the data onto a randomly chosen unit direction and then splits the set into two roughly equal-sized sets using new split rules. Dasgupta and Freund [6] have proposed two rules for RP-trees: RP-tree max and RP-tree mean. The difference between the two rules is that RP-tree mean occasionally performs a different kind of split based on the distance from the mean of the coordinates. RP-tree has been used in many other applications as a better alternative to K-means [32].

4.1.2 Tree Construction

We use RP-tree max or RP-tree mean rule to partition the input large dataset into many small clusters. In practice, we observe that RP-tree mean rule would compute k-nearest neighbor results with better quality in terms of the recall of the overall bi-level scheme.

To apply RP-tree mean rule, we need to efficiently compute $\Delta(S)$, the diameters for a given point set S in a high-dimensional space, which is in fact as difficult as the original k-nearest neighbor query. Fortunately, there exist algorithms that can approximate the diameter efficiently. We use the iterative method proposed by Egecioglu and Kalantari [10], which converges fast to a diameter approximation with good precision. Moreover, this approach can be easily parallelized, which is important for our bi-level scheme's performance on large datasets [23]. The diameter computation algorithm uses a series of m values $r_1, ..., r_m$ to approximate the diameter for a point set S, where $m \leq |S|$. It can be proved that $r_1 < r_2 < ... < r_m \leq \Delta(S) \leq \min(\sqrt{3}r_1, \sqrt{5-2\sqrt{3}}r_m)$ [10]. In practice, we find that r_m is usually a good enough approximation of $\Delta(S)$ even when m is small (e.g. 40). The time complexity of this approximate diameter algorithm is $O(m|S|)$ and we can construct each level of the RP-tree in time that is linear in the size of the entire dataset. As a result, the overall complexity to partition the dataset into g groups is $O(\log(g)n)$.

4.1.3 Analysis

The RP-tree construction algorithm is simple. However, we show that it can improve the quality of k-nearest neighbor queries and reduce the performance variations in the overall bi-level scheme.

One of our main goals to use RP-tree is to enable our LSH scheme to adapt to datasets with different distributions and to obtain higher quality and lower runtime cost for a given dataset. The RP-tree partitions a dataset into several leaf nodes so that each leaf node only contains similar data items, e.g. images for the same or similar objects, which are likely to be drawn from the same distribution. During the second level of our Bi-level LSH scheme, we may choose different LSH parameters (e.g. bucket size W) that are optimal for each cell instead of choosing a single set of parameters for the entire dataset. Such parameter choosing strategy can better capture the interior differences within a large dataset.

Kd-tree and K-means can also perform partitioning on data items, but RP-tree has some advantages over these methods for high-dimensional datasets. An ideal partitioning algorithm should have fast convergence speed and should be able to adapt to data's intrinsic dimension. In our case, the convergence speed is measured by the number of tree levels needed to halve the radius of one RP-tree node. Moreover, the intrinsic dimension represents the fact that usually the data items are on a low-dimensional submanifold embedded in the high dimensional Euclidean space. Two widely used measures of intrinsic dimension are the *Assouad dimension* and the *local covariance dimension* [6].

For a dataset with intrinsic dimension d, which is usually much smaller than D, an ideal partitioning algorithm should have high convergence speed. However, it is known that in the worst case, the Kd-tree would require $\mathcal{O}(D)$ levels so as to halve the radius of cells in $\mathbb{R}^D$ even when $d \ll D$, which is slow. One possible way to make Kd-tree adaptive to intrinsic dimension is to project the data to the direction with maximum spread, which can be computed by PCA, instead of the coordinate directions. However, PCA is time-consuming on high-dimensional data and does not work well on datasets with multiple clusters. K-means is another data-partitioning approach, but it is sensitive to parameter initialization and may be slow to converge to a good solution.

As compared to these methods, RP-tree has faster and guaranteed convergence speed and can adapt to intrinsic dimension automatically, according to the properties of RP-tree max and RP-tree mean (taken from [6, 8]):

THEOREM 1. *Suppose S's Assouad dimension $\leq d$, in the worst case RP-tree max rule can halve the cell radius in $\mathcal{O}(d \lg d)$ levels with high probability.*

THEOREM 2. *Suppose S's local covariance dimension is (d, ϵ, r), on expectation, RP-tree mean rule will reduce the cell radius from r to $c \cdot r$ in each level of the tree, where $c < 1$ and depends on d and ϵ. The randomization is over the construction of RP-tree as well as different choice of data points.*

Intuitively, the RP-tree max converges quickly in terms of the worst-case improvement (i.e. the decreasing of cell radius) amortized over levels while the RP-tree mean converges quickly in terms of the average improvement during each level.

Another main advantage of RP-tree is that it can reduce the performance and quality variation of LSH algorithms. Previous statistical models related to LSH, such as [22, 9, 15], are based on optimizing the average performance and quality over all the random sampled projections. However, the actual performance and quality of the resulting algorithms may deviate from the mean values according to the magnitude of the variance caused by random projections, which is related to the 'shape' of the dataset. Let us consider the LSH scheme on l_p (i.e. Equation (2)) as an example. First, notice that the length of projection vector

$\|\mathbf{a}_i\|^2 \sim \chi^2_D \approx N(D, 2D)$ is approximately a constant as $\sqrt{2D} \ll D$. Therefore, we here assume the projection vector $\mathbf{a}_i$ as a unit direction, because otherwise we can scale both $\mathbf{a}_i$ and W by $\frac{1}{\sqrt{D}}$ and this scaling will not change the LSH function in Equation (2). In this case, we can show that the shape of the dataset will influence the variance in performance and quality of the LSH algorithm. As shown in Figure 2(a), if the dataset shape is flat, then the projection along the long axis will result in a hash function with small recall ratio but also small selectivity (i.e. small runtime cost) while the projection along the short axis will result in a hash function with large recall ratio but large selectivity (i.e. large runtime cost). It is very hard to compute a value of W that is suitable for all the projections in terms of both quality and runtime cost. Therefore, we can have large variance in performance or quality. Instead, if the shape is nearly 'round', as shown in Figure 2(b), then the recall ratio and the runtime cost would be similar along all projection directions and the deviation in overall performance or quality is small. The dataset shape's influence on LSH scheme can also be observed based on analyzing locality-sensitive hash functions. The probability that two data items are located in the same interval when projected onto one random direction $\mathbf{a}_i$ is $\mathbb{P}[h_i(\mathbf{u}) = h_i(\mathbf{v})] = 1 - \frac{\min(d,W)}{W}$, where $d = |\mathbf{a}_i \cdot \mathbf{u} - \mathbf{a}_i \cdot \mathbf{v}|$. Apparently, if $\mathbf{a}_i$ is along the long axis of a dataset, then $\mathbb{P}[h_i(\mathbf{u}) = h_i(\mathbf{v})]$ will be small for most $(\mathbf{u}, \mathbf{v})$ pairs, which results in low recall ratio and selectivity. Otherwise, $\mathbf{a}_i$ along the short axis will result in high recall ratio and selectivity. Even when using multi-probe LSH techniques, performance variance still exists among different projections because the probability that two points fall in the intervals that are e steps away is $\mathbb{P}[|h_i(\mathbf{u}) = h_i(\mathbf{v})| \leq e] = 1 - \frac{\max[\min(d-e\cdot W, W), 0]}{W}$, which is also a decreasing function of d.

As a result, LSH algorithms work better on datasets with a 'round' shape or with bounded aspect ratio [8]. RP-tree max rule is proved to be able to partition a dataset into subsets with bounded aspect ratios and therefore reduces the performance variance of LSH algorithms:

THEOREM 3. *(taken from [8]) Under suitable assumptions, there is a high probability that RP-tree max rule will produce cells with aspect ratio no more than $\mathcal{O}(d\sqrt{d}\lg d)$, where d is the cell's Assouad dimension.*

Bounded aspect ratio is more difficult to prove for RP-tree mean rule. However, from Theorem 2, after each level the volume of the dataset's cover ball is reduced to c^D times the original ball, but the dataset size is only halved. Therefore, the average density for each subset increases quickly after each level, which usually implies smaller aspect ratio.

4.2 Level II: Hierarchical LSH

In the second level of the bi-level algorithm, we construct LSH hash table for each leaf node cell calculated during the first level. Before that, we use an automatic parameter tuning approach [9] to compute the optimal LSH parameters for each cell. The RP-tree computed during the first level has already partitioned the dataset into clusters with nearly homogeneous properties. As a result, the estimated parameters for each cluster can improve the runtime performance and quality of the LSH scheme, as compared to the single set of parameters used for the entire dataset. The LSH code of one item and its leaf node index composes its Bi-level LSH code, i.e. $\tilde{H}(\mathbf{v}) = (\text{RP-tree}(\mathbf{v}), H(\mathbf{v}))$.

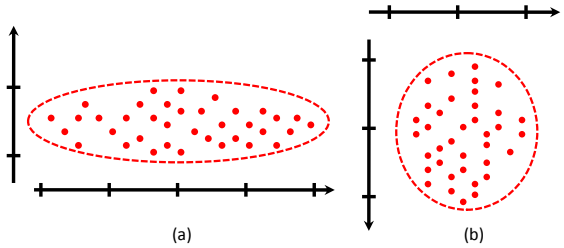


Figure 2: LSH scheme behaves better on dataset with bounded aspect ratio. (a) The data has a large aspect ratio and there is no bucket size W that is suitable for all random projections. (b) The dataset has a shape close to a sphere and W chosen for one projection is usually suitable for all the projections.

The LSH table is implemented as a linear array along with an indexing table. The linear array contains the Bi-level LSH codes of all the items in the dataset, which have been sorted to collect items with same LSH codes together. All the data items with the same LSH code constitute a bucket which is described by the start and end positions of the bucket (i.e. the bucket interval) in the sorted linear array. As each bucket uniquely corresponds to one LSH code, we can use the terminology 'bucket' and 'unique LSH code' in the same sense. The indexing table corresponds to a cuckoo hash table with each key as one LSH code and the value associated with the key is the corresponding bucket interval. Another important reason why we implement LSH table in this way is that the cuckoo hash table can be parallelized on current commodity processors [23].

In order to make LSH approach adaptive to different queries, we further enhance the LSH table with a hierarchical structure. When the given query lies in a region with low data density (e.g. a region between two clusters), basic LSH or multi-probe techniques may have only a few hits in the LSH hash table and therefore, there may not be a sufficient number of neighborhood candidates for a given k-nearest neighbor query. As a result, the recall ratio could be small. In such cases, the algorithm should automatically search the far away buckets or larger buckets so as to improve multi-probe's efficiency and ensure that there are sufficient numbers of neighborhood candidates. Our solution is to construct a hierarchy on the buckets. Given a query code, we can determine the hierarchical level that it lies in and provide the suitable buckets for the probing algorithm.

For the $\mathbb{Z}^M$ lattice, we implement the hierarchy using the space filling Morton curve, also known as the Lebesgue or Z-order curve. We first compute the Morton code for each unique LSH code by interleaving the binary representations of its coordinate value and then sort the Morton codes to generate a one-dimensional curve, as shown in Figure 4(a). The Morton curve maps the multi-dimensional data to a single dimension and tries to maintain the neighborhood relationship, i.e. nearby points in the high-dimensional space are likely to be close on the one-dimensional curve. Conceptually, the Morton curve can be constructed by recursively dividing dim-D cube into 2^D cubes and then ordering the cubes, until at most one point resides in each cube. Note that the Morton code has one-to-one relationship with the LSH code, so it can also be used as the key to refer to the

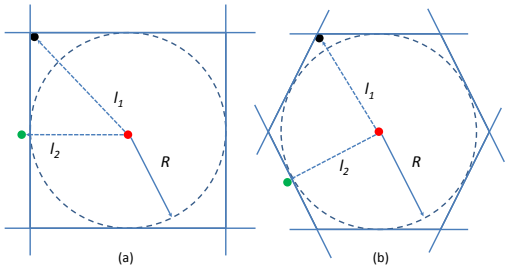


Figure 3: Given a query (the red point), lattice with higher density may provide recall ratio with lower selectivity. (a) is the $\mathbb{Z}^M$ lattice and $\sqrt{M}R = l_1 \gg l_2 \approx R$ if M is large. If points are distributed uniformly in the cube cell, almost all points inside the cube are outside the inscribed ball like the black point and therefore are poorer neighborhood candidates than the green point outside the cell. (b) is a lattice with higher density and $l_1 \approx l_2 \approx R$ and all the points in the cell are good neighborhood candidates.

buckets in the LSH hash table.

Suppose that we have constructed the LSH Morton curve for the given dataset. Given one query item, we first compute its LSH code and the corresponding Morton code. Next, we search within the Morton curve to find the position where the query code can be inserted without violating the ordering. Finally, we use the Morton codes before and after the insert position in the Morton curve to refer the buckets in the LSH hash table for short-list search. As the Morton curve cannot completely maintain the neighborhood relationship in high-dimensional space, we need to perturb some bits of the query Morton code and repeat the above process several times [21]. We can also compute the number of most significant bits (MSB) shared by query Morton code and the Morton codes before and after the insert position. When the shared MSB number is small, we should traverse to a higher level in the hierarchy and use a large bucket, which is implemented as buckets with the same MSB bits.

To overcome the curse of dimensionality difficulty caused by basic $\mathbb{Z}^M$ lattice, we also improve the hierarchical LSH table using lattice techniques. The drawback of $\mathbb{Z}^M$ lattice is that it has low density in high-dimensional spaces and cannot provide high quality nearest neighbor candidates for a given query: most candidates inside the same bucket with the query item may be very far from it and many of the actual nearest neighbors may be outside the bucket. For example, as shown in Figure 3(a), there exist points in one cell that has the same distance to the center of cell with points in cells $\sqrt{D}$ steps away. As a result, we may need to probe many cells to guarantee large enough recall ratio, which implies large selectivity. One solution to this drawback is using E_8 lattice [14] which has the maximum density in dim-8 space. Intuitively, E_8 lattice, as shown in Figure 3(b), is closest to dim-8 sphere in shape and thus can provide high quality nearest neighbor candidates. We do not use the Leech lattice [1] because the E_8 lattice has more efficient decode algorithm.

The E_8 lattice is the collection of points in dim-8 space, whose coordinates are all integers or are all half-integers and the sum of the eight coordinates is an even integer, e.g., $(1)^8 \in E_8$, $(0.5)^8 \in E_8$ but $(0,1,1,1,1,1,1,1) \notin E_8$. The collection of all integer points whose sum is even is called the D_8 lattice and we have $E_8 = D_8 \cup (D_8 + (\frac{1}{2})^8)$. One main difference between $\mathbb{Z}^8$ and E_8 lattice is that each E_8 lattice node has 240 neighbors with the same distance to it and requires a different multi-probe and hierarchy construction strategy as compared to $\mathbb{Z}^M$ lattice.

For convenience, we first assume the space dimension to be 8. Then given a point $\mathbf{v} \in \mathbb{R}^8$, its E_8 LSH code is

$$H_{E_8}(\mathbf{v}) = \textsc{Decode}\left([\frac{\mathbf{a}_1 \cdot \mathbf{v} + b_1}{W}, \frac{\mathbf{a}_2 \cdot \mathbf{v} + b_2}{W} ..., \frac{\mathbf{a}_8 \cdot \mathbf{v} + b_8}{W}]\right). \tag{6}$$

That is, we replace the floor function in basic LSH hash function by the E_8 decode function $\textsc{Decode}(\cdot)$, which maps the vector in $\mathbb{R}^8$ onto E_8 lattice points. There exists an efficient implementation of the E_8 decoder [14] with only 104 operations.

The multi-probe process for the E_8 LSH algorithm is as follows. Given a query, we first probe the bucket that it lies in and then the 240 buckets closest to it. The probe sequence is decided by the distance of query to the 240 E_8 lattice nodes, similar to the case in $\mathbb{Z}^8$ [22]. If the number of candidates collected is not enough, we recursively probe the adjacent buckets of the 240 probed buckets.

The 240 neighborhood property makes it more complicated to construct hierarchy for the E_8 lattice. Morton code is not suitable for the E_8 lattice because it needs orthogonal lattices like $\mathbb{Z}^M$. However, Morton curve can also be viewed as an extension of octree in $\mathbb{Z}^M$: partition a cube by recursively subdividing into 2^M smaller cubes. The subdivision is feasible because $\mathbb{Z}^M$ has the *scaling* property: the integer scaling of $\mathbb{Z}^M$ lattice is still a valid $\mathbb{Z}^M$ lattice. Suppose $\mathbf{c} = (c_1, ..., c_M) = H(\mathbf{v}) \in \mathbb{R}^M$ is the traditional $\mathbb{Z}^M$ LSH code for a given point $\mathbf{v}$, then its k-th ancestor in the LSH code hierarchy is

$$H^k(\mathbf{v}) = 2^k\big(\underbrace{\lfloor\frac{1}{2}...\lfloor\frac{1}{2}}_{k}\lfloor\frac{c_1}{2}\rfloor\rfloor...\rfloor, ..., \underbrace{\lfloor\frac{1}{2}...\lfloor\frac{1}{2}}_{k}\lfloor\frac{c_M}{2}\rfloor\rfloor...\rfloor\big) \tag{7}$$

$$= 2^k\big(\lfloor\frac{c_1}{2^k}\rfloor, ..., \lfloor\frac{c_M}{2^k}\rfloor\big), \tag{8}$$

where $k \in \mathbb{Z}^+$ and $H^0(\mathbf{v}) = H(\mathbf{v})$. The second equality is due to

$$\lfloor\frac{1}{n}\lfloor\frac{x}{m}\rfloor\rfloor = \lfloor\frac{x}{mn}\rfloor \tag{9}$$

for all $m, n \in \mathbb{Z}$ and $x \in \mathbb{R}$. E_8 lattice also has the scaling property, so we can construct the hierarchy in E_8 by defining the k-th ancestor of a point $\mathbf{v} \in \mathbb{R}^8$ as

$$H^k_{E_8}(\mathbf{v}) = 2^k\big(\underbrace{\textsc{Decode}(\frac{1}{2}...\,\textsc{Decode}(\frac{1}{2}\,\textsc{Decode}(\frac{c_1}{2}))...)}_{k},$$
$$\underbrace{..., \textsc{Decode}(\frac{1}{2}...\,\textsc{Decode}(\frac{1}{2}\,\textsc{Decode}(\frac{c_M}{2}))...)}_{k}\big) \tag{10}$$

where $k \in \mathbb{Z}^+$ and $\mathbf{c} = H^0_{E_8}(\mathbf{v}) = H_{E_8}(\mathbf{v})$. The decode function does not satisfy the same property as Equation (9) for the floor function, so we cannot simplify it to have a form similar to Equation (8).

There is no compact representation of E_8 LSH hierarchy similar to the Morton curve for $\mathbb{Z}^M$ lattice. We implement the E_8 hierarchy as a linear array along with an index hierarchy, as shown in Figure 4(b). 1) First, we generate a

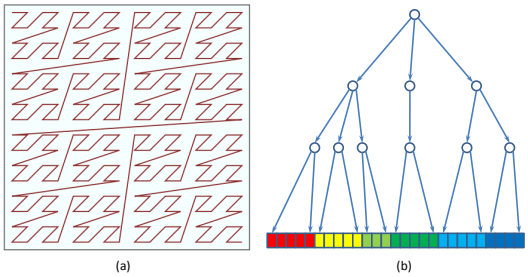


Figure 4: (a) $\mathbb{Z}^M$ lattice uses the Mortion curve as the hierarchy, which can keep the neighborhood relationship in a one dimensional curve. (b) E_8 lattice's hierarchy is a tree structure which better fits E_8 lattice's 240 neighborhood property. In each level of the hierarchy, points with the same LSH code at that level are collected together.

set by selecting one point from each E_8 bucket. The linear array starts with the E_8 LSH codes $H^m_{E_8}(\mathbf{v})$ for all the points in the set, where m is the smallest integer that ensures all items have the same code. 2) Then we compute $H^{m-1}_{E_8}(\mathbf{v})$ for all points and group items with the same code together using sorting. Each hierarchy tree node stores the start and end position for one subgroup and its common E_8 code. The process is repeated until $m = 0$ and the linear array contains the common E_8 code for each bucket. Given a query, we traverse the hierarchy in a recursive manner by visiting the child node whose E_8 code is the same as the query at that level. The traversal stops until such a child node does not exist and all the buckets rooted from current node need to be searched for the multi-probe process.

If the dimension of the dataset is $M > 8$, we use the combination of $\lceil \frac{M}{8} \rceil$ E_8 lattices to represent the LSH table.

5. ANALYSIS

One main advantage of our bi-level scheme is that it can generate more compact LSH code than prior methods. In our case, the compactness means that we can use a LSH code with fewer bits to generate results with similar recall ratio/selectivity. According to [31], a compact LSH code should have the property that the variance of each bit is maximized and the bits are pairwise uncorrelated.

The compactness of Bi-level LSH depends on the following property of random projections [11]: when perform clustering in a high-dimensional space, a random direction is almost as good as the principle eigenvector to identify the subspaces (i.e. clusters) in the data. Also notice that LSH scheme can be viewed as a special case of random projection tree where each RP-tree node uses the same projection direction and generates one bit of LSH code, as shown in Figure 5(a)(b). As a result, the high-order bits generated by different LSH projection directions are highly correlated and there exists considerable redundancy in the codes. Such a conclusion can also be verified by results reported by previous work about locality-sensitive coding [31, 25]. The hash family used in [31] is $h_{ij}(\mathbf{v}) = \text{sgn}[\sin(\frac{\pi}{2} + \frac{j\pi}{b-a}\mathbf{a}_i \cdot \mathbf{v})]$, where $j \in \mathbb{Z}^+$ and $\mathbf{a}_i$ are the principal directions computed using PCA. Moreover, the hash family used in [25] is $h_i(\mathbf{v}) = \frac{1}{2}\{1+\text{sgn}[\cos(\mathbf{a}_i \cdot \mathbf{v}+b_i)+t_i]\}$, where $\mathbf{a}_i \sim N(0, \gamma\mathbf{I})$, $t_i \sim U[-1, 1]$ and $b_i \sim U[0, 2\pi]$ are independent of one another. As shown in Figure 5(c)(d), the two approaches only use the lowest bits of LSH code, but when using codes of the same length, they can generate results with higher recall and lower selectivity than standard LSH code [7]. Our Bi-level LSH scheme removes the redundancy in the high-order bits by decomposing LSH hashing function $H(\mathbf{v})$ into $\tilde{H}(\mathbf{v}) = \big(\text{RP-tree}(\mathbf{v}), H(\mathbf{v})\big)$, where the same RP-tree is shared by all M hashing functions in $H(\cdot)$. In other words, the Bi-level LSH code is generated in two steps: The RP-tree is the first step to generate high-order bits, while traditional LSH scheme is used to generate the lower-order bits. Moreover, the bi-level scheme also results in increasing the variance of the high-order bits (i.e. RP-tree($\mathbf{v}$)) of the resulting LSH code because in practice, the RP-tree can partition the dataset into subsets with similar sizes.

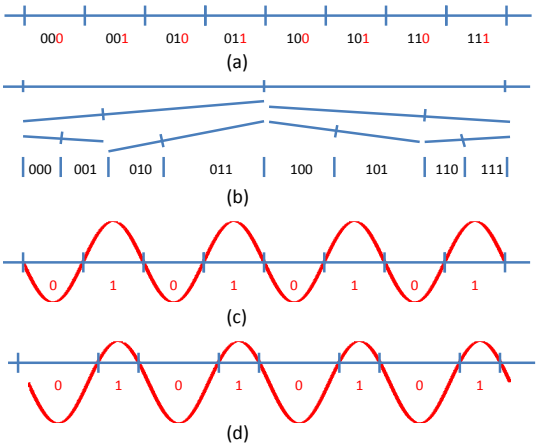


Figure 5: Hash codes generated by different LSH methods: (a) basic LSH [11] (b) random projection tree [11, 6] (c) Spectral hashing [31] (d) locality-sensitive binary codes [25]. (a)(c)(d) are data-independent and (c)(d) only use the lowest-order bits of (a). (b) is similar to (a) but uses different project directions to generate each bit.

6. IMPLEMENTATION AND RESULTS

In this section, we compare the performance of our Bi-level LSH algorithm with prior LSH methods. All the experiments are performed on a PC with an Intel Core i7 3.2GHz CPU and NVIDIA GTX 480 GPU. The system has 2GB memory and 1GB video memory.

6.1 Datasets

We employ two datasets to evaluate our method. One benchmark is the LabelMe image dataset (`http://labelme.csail.mit.edu`), which includes nearly 200,000 images. Each image is represented as a GIST feature of dimension 512. The other benchmark is the Tiny Image database (`http://horatio.cs.nyu.edu/mit/tiny/data/index.html`), which has 80 million images. Each image is represented as a GIST feature of dimension 384.

6.2 Quality Comparison

6.2.1 *Experiment Design*

We compare the quality of our bi-level LSH scheme with prior LSH methods in several ways. First, given the same se-

lectivity (Equation (5)), we compare the recall ratio (Equation (3)) and error ratio (Equation (4)) of different methods. The selectivity measures the number of potential candidates for short-list search and is proportional to the algorithm's runtime cost. Recall and error ratio measure how close the approximate result is to the accurate result. Therefore, this criteron means that given the same runtime budget, we should prefer the algorithm that gives the best quality. Second, we compare the standard deviations due to randomly selected projections between prior LSH approaches and our bi-level scheme. The LSH-based methods are randomized algorithms whose performance may change when the projection vectors are chosen randomly. In our case, we tend to ensure that the algorithm gives high quality results with low runtime cost even in the worst case, and not the average case over random projections. Finally, we compare the standard deviations of different methods with our approach over different queries. Our goal is to ensure that the algorithm computes k-nearest neighbor results with similar quality and runtime cost for all different queries.

In terms of the comparison, we use 100,000 items in the dataset to calculate the LSH hash table and then use another 100,000 items in the same dataset as the queries to perform k-nearest neighbor search using different LSH algorithms. For one specific LSH algorithm, we use three different L values (10, 20, 30) and for each L, we increase the bucket size W gradually which will result in selectivities in an ascending order. We keep all other parameters fixed ($M = 8$, $k = 500$). For each W, we execute the 100,000 k-nearest neighbor queries 10 times with different random projections. For each execution of the LSH algorithm, we compute its recall ratio (ρ), error ratio (κ) and selectivity (τ) as the measurements for quality and runtime cost, which change along the bucket size W. According to our previous analysis in Section 4.1.3, the three measurements are random variables and we have $\rho = \rho(W, r_1, r_2)$, $\kappa = \kappa(W, r_1, r_2)$ and $\tau = \tau(W, r_1, r_2)$, where r_1 and r_2 are two random variables representing various projections and queries, respectively.

In order to compare the quality among different methods given the same selectivity, we need to compute two curves: $\rho(W) = f(\tau(W))$, which describes the relationship between selectivity and recall ratio and $\kappa(W) = g(\tau(W))$, which describes the relationship between selectivity and error ratio. We estimate the two curves based on expected measurements over all LSH executions with the same W but different projection directions and queries

$$\mathbb{E}_{r_1,r_2}(\rho(W)) = \tilde{f}(\mathbb{E}_{r_1,r_2}(\tau(W))) \tag{11}$$

and

$$\mathbb{E}_{r_1,r_2}(\kappa(W)) = \tilde{g}(\mathbb{E}_{r_1,r_2}(\tau(W))). \tag{12}$$

$\tilde{f}$ and $\tilde{g}$ are in fact conservative estimations of the quality because

$$\tilde{f}(\mathbb{E}_{r_1,r_2}(\tau)) = \mathbb{E}_{r_1,r_2}(\rho) = \mathbb{E}_{r_1,r_2}(f(\tau)) \leq f(\mathbb{E}_{r_1,r_2}(\tau))$$

and therefore $\tilde{f} \leq f$. Here $\leq$ is due to the fact that f is usually a concave function. Similarily, we have $\tilde{g} \leq g$. Moreover, We use standard deviations $\mathrm{Std}_{r_1}(\mathbb{E}_{r_2}(\tau))$, $\mathrm{Std}_{r_1}(\mathbb{E}_{r_2}(\rho))$, $\mathrm{Std}_{r_1}(\mathbb{E}_{r_2}(\kappa))$ to measure the diversity of selectivity, recall ratio and error ratio caused by projection randomness. Moreover, we use the standard deviations $\mathrm{Std}_{r_2}(\mathbb{E}_{r_1}(\tau))$, $\mathrm{Std}_{r_2}(\mathbb{E}_{r_1}(\rho))$, $\mathrm{Std}_{r_2}(\mathbb{E}_{r_1}(\kappa))$ to measure the diversity of selectivity, recall ratio and error ratio caused by query diversity. In practice, we use arithmetic mean of the measurements to approximate the expectation of the measurements, assuming the project directions and the queries are sampled uniformly.

There are several main differences between our experiment and those performed in prior approaches such as [22] and [9]. First, we pay more attention to the variances due to randomness of LSH scheme as one of our main goals is to reduce these variances. Secondly, we search a neighborhood with larger size ($k = 500$) than the $k = 20$ or 50 used in previous works [22, 9], which increases the difficulty to obtain high recall ratio. Finally, we use a large query set ($100,000$) instead of the several hundred queries used in previous methods, which helps to estimate the LSH algorithms' quality and runtime cost in real world data. The large neighborhood size, query number and the repetition with different random projections require more computations in our experiments, as compared to the others reported in prior work. We also use GPU parallelism for the fast computation [23], which provides more than 40 times acceleration over a single-core CPU implementation.

6.2.2 Comparison Results

The LSH algorithms that we compared include standard LSH and its two variations using multiprobe or hierarchy technique (i.e. multiprobed standard LSH and hierarchical standard LSH), Bi-level LSH and its two variations using multiprobe or hierarchy technique (i.e. multiprobed Bi-level LSH and hierarchical Bi-level LSH). For each LSH algorithm, we further test its performance when using $\mathbb{Z}^M$ and E_8 lattice, respectively.

First, we compare the standard LSH algorithm with our Bi-level LSH algorithm while using $\mathbb{Z}^M$ lattice as the space quantizer, as shown in Figure 6. We notice the following facts from the results: 1) Given the same selectivity, our bi-level scheme can usually provide higher recall ratio and error ratio than standard LSH algorithm. The reason is that the RP-tree partition in the first level clusters similar feature together, which are more likely to be close to each other in the feature space. This partition provides better locality coding than the standard LSH hash function. However, we also observe that when the selectivity is large (0.6 in Figure 6(b) and 0.5 in Figure 6(c)), the limit recall ratio of Bi-level LSH can be smaller than that of standard LSH. The reason is that given the same bucket size W, Bi-level LSH always has more buckets than standard LSH due to the first level partition in bi-level scheme. However, in practice we are only interested in selectivity less than 0.4, otherwise the efficiency benefit of LSH over brute-force method is quite limited. We also observe that Bi-level LSH always provides results that are better than standard LSH. 2) We observe that when the selectivity increases, the error ratio increases much faster than the recall ratio. This is due to the concentration effect mentioned earlier. 3) Our Bi-level LSH has smaller standard deviation due to projection randomness than standard LSH (i.e. smaller ellipse in the figures), for both quality and selectivity. 3) When selectivity is small (i.e. small W), the standard deviation of quality is large while the standard deviation of selectivity is small. When selectivity is large (i.e. large W), the standard deviation of selectivity is large while the standard deviation of quality is small. The reason is that for small W, different projections will produce results with very different quality, while for large

W, the quality (i.e. recall and error ratio) of the k-nearest neighbor results will converge to 1 whatever projections are selected. 4) When L increases, the standard deviations of both Bi-level LSH and standard LSH decreases. The reason is that the probability that two points $\mathbf{u}$, $\mathbf{v}$ are projected into the same bucket by one of the L random projections is $1-\prod_{i=1}^{L}(1-\mathbb{P}[h_i(\mathbf{u})=h_i(\mathbf{v})]) = 1-\prod_{i=1}^{L}\frac{\min(|\mathbf{a}_i\cdot\mathbf{u}-\mathbf{a}_i\cdot\mathbf{v}|,W)}{W}$. Therefore, the final probability is related with the geometric average of the probabilities when using different projections and will converge to 1 quickly. However, we can see that the standard deviations of Bi-level LSH when using $L = 10$ is almost as small as the standard deviations of standard LSH when using $L = 20$ or 30, which means that we save 2/3 runtime cost and memory for LSH hash tables when using our Bi-level LSH to obtain similar quality with standard LSH.

Figure 7 shows the comparison between the standard LSH algorithm and our Bi-level LSH algorithm while using E_8 lattice as the space quantizer. We can see that the results are similar to the case when using $\mathbb{Z}^M$ lattice, though E_8 lattice offers better performance at times. Bi-level LSH also outperforms standard LSH in this case.

Figure 8 and Figure 9 show the comparison between Bi-level LSH and standard LSH when both methods are enhanced with the multiprobe technique [22], using $\mathbb{Z}^M$ and E_8 lattice respectively. We use 240 probes for each methods. For $\mathbb{Z}^M$ lattice, we use heap-based method in [22] to compute the optimal search order for each query. For E_8 lattice, we simply use the 240 neighbors of the lattice node that one query belongs to. As in previous cases, we observe that the Bi-level LSH results in better quality as compared to standard LSH. Figure 12 and Figure 13 compare the selectivity-recall curve among different methods when using $\mathbb{Z}^M$ lattice or E_8 lattice respectively. We can see that for $\mathbb{Z}^M$ lattice, the quality when using multiprobe is better than the quality when not using multiprobe, for both Bi-level LSH and standard LSH. While for E_8 lattice, the quality when not using multiprobe is slightly better than the quality that is obtained using multiprobe. The reason is that multiprobe is mainly used to obtain better quality with fewer LSH hash tables (i.e. L) by probing nearby buckets besides the bucket that the query lies in. However, these additional buckets may have a smaller probability to contain the actual k-nearest neighbors than the bucket that the query lies in and therefore require much more selectivity in order to obtain small improvements in quality. E_8 lattice has higher density than $\mathbb{Z}^M$ lattice and therefore as compared to $\mathbb{Z}^M$ lattice, the additional E_8 buckets to be probed will contain more potential elements for short-list search but only a few of them could be the actual k-nearest neighbors. As a result, multiprobe technique has a larger performance degradation on E_8 lattice than on $\mathbb{Z}^M$ lattice. However, we observe that the multiprobe technique can reduce the variance caused by projection randomness. The reason is that multiprobe technique is equivalent to using larger L, which we have explained to be able to reduce deviation caused by projection randomness.

Figure 10 and Figure 11 show the comparison between Bi-level LSH and standard LSH when both methods are enhanced with the hierarchical structure introduced in Section 4.2, using $\mathbb{Z}^M$ and E_8 lattice respectively. Given a set of queries, we first compute the potential elements for short-list search using Bi-level LSH or standard LSH. Next, we compute the median of all queries' short-list size. For those queries with short-list size smaller than the median, we search the LSH table hierarchy to find suitable bucket whose size is larger than median. The hierarchical strategy is used to reduce the deviation caused by query diversity. However, from Figure 10 and Figure 11 we find that it can also help to reduce the deviation caused by random projections. Moreover, as shown in Figure 12 and Figure 13, the hierarchical strategy will not cause the quality degradation similar to multiprobe technique. Still, Bi-level LSH performs better than standard LSH.

We compare the selectivity-recall ratio among all the methods in Figure 12 and Figure 13. For $\mathbb{Z}^M$ lattice, multiprobed Bi-level LSH provides the best recall ratio. Moreover, Bi-level LSH, hierarchical Bi-level LSH and multiprobed standard LSH provide similar recall ratios. In contrast, standard LSH and hierarchical standard LSH have the worst recall ratios. For $\mathbb{E}_8$ lattice, Bi-level LSH, hierarchical Bi-level LSH and multiprobed Bi-level LSH provide the best recall ratios. Standard LSH and hierarchical standard LSH have similar quality. The multiprobed standard LSH has the worst quality. We also use Figure 12 and Figure 13 to compare the deviation caused by query diversity between different methods. We can see that the hierarchical Bi-level LSH provides results with the smallest deviation among all the methods. Hierarchical standard LSH also provides results with smaller deviation than standard LSH and multiprobed standard LSH.

6.3 Parameter Comparison

We now compare the Bi-level LSH when using different parameters. In all comparisons, we use $L = 20$.

Figure 14(a) shows the results when Bi-level LSH uses different number of partitions (1, 8, 16, 32, 64) in the first level. We can observe that when group number increases, the quality increases given the same number of selectivity. However, this increase slows down after 32 partitions.

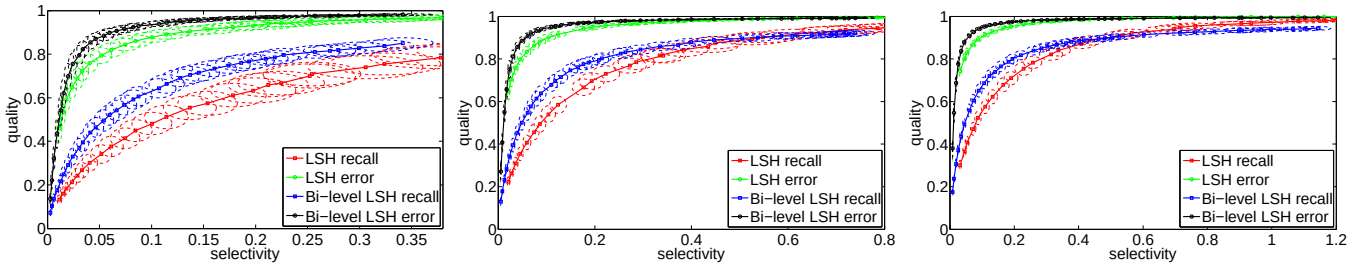
Figure 14(b) compares Bi-level LSH with standard LSH with different setting for M. Bi-level LSH uses the hash code $\tilde{H}(\mathbf{v}) = \big(\text{RP-tree}(\mathbf{v}), H(\mathbf{v})\big)$. The first level adds some additional code before the standard LSH code. This comparison shows that the improvement of Bi-level LSH is due to using better code but not longer code. As shown in the result, Bi-level LSH provides better quality than standard LSH using different M.

Figure 14(c) compares the Bi-level LSH when using K-means and RP-tree in the first level. We can see that the quality and deviation when using RP-tree is better than those when using K-means.

7. CONCLUSION AND FUTURE WORK

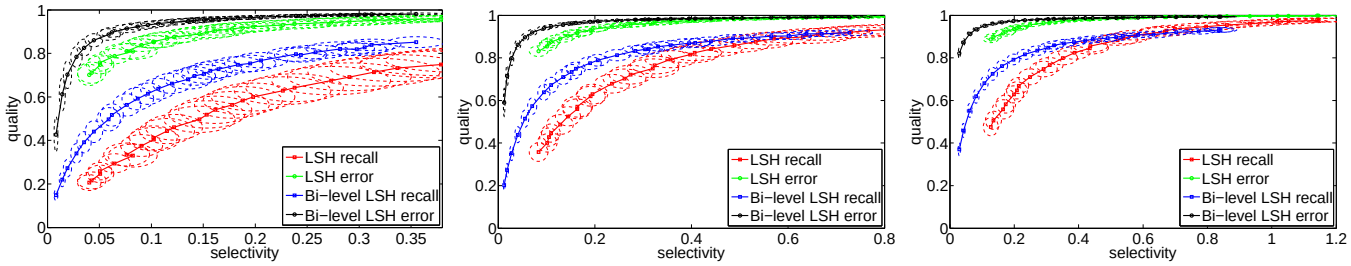
We have presented a new Bi-level LSH based algorithm for efficient k-nearest neighbor search. We use RP-tree to reduce the variation caused by randomness in LSH framework, to make hashing adaptive to datasets and improve the compactness of LSH coding. We also construct a hierarchy on LSH tables so as to make the algorithm adaptive to different queries. The hierarchy is also enhanced by E_8 lattice to handle the curse of dimensionality problem. In practice, our algorithm can compute k-nearest neighbor results with improved quality as compared to prior LSH methods when given the same selecitivity budget and can generate results with less variance in quality and runtime cost.

There are many avenues for future work. We hope to test



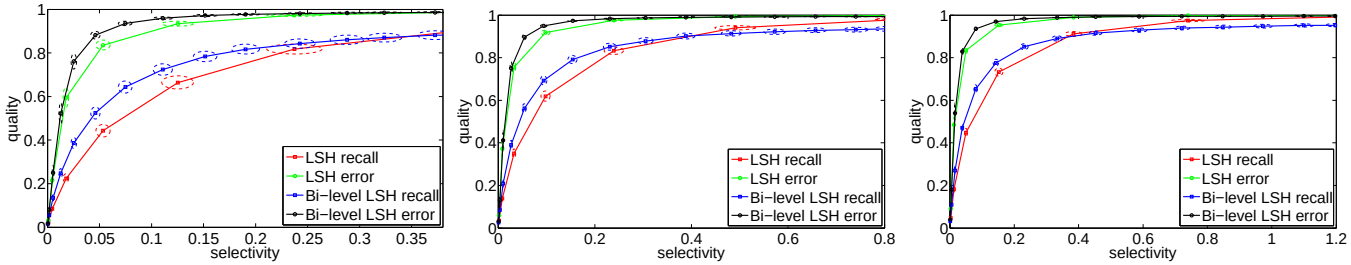
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 6: Quality comparison between our Bi-level LSH and standard LSH when using $\mathbb{Z}^M$ lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



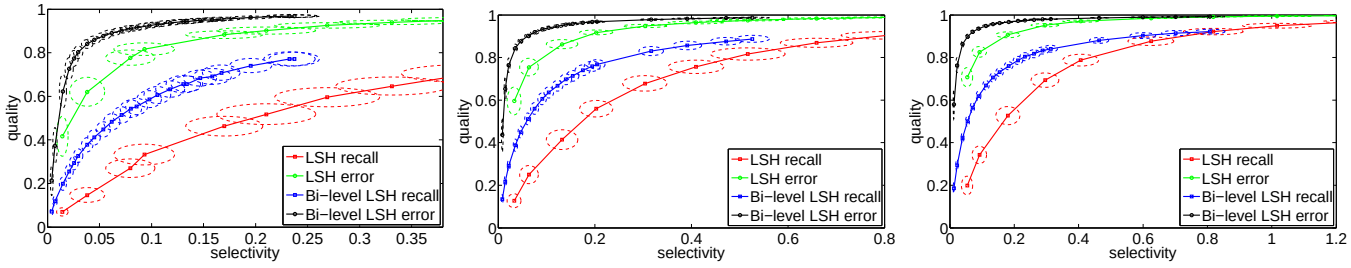
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 7: Quality comparison between our Bi-level LSH and standard LSH when using E_8 lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



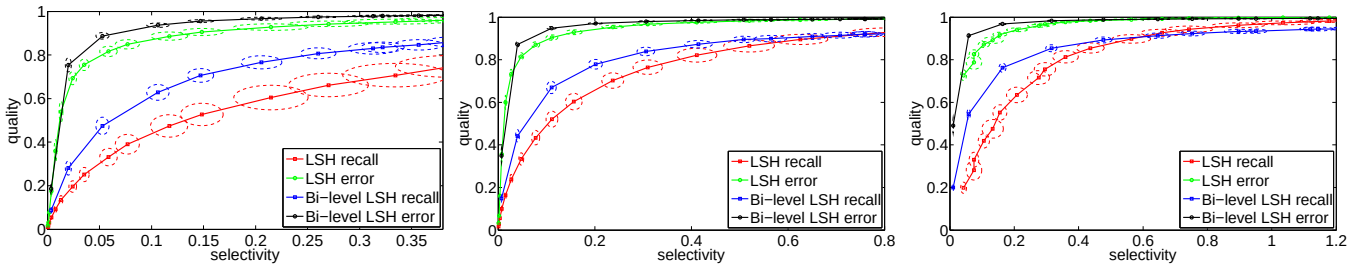
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 8: Quality comparison between our multiprobed Bi-level LSH and standard multiprobed LSH when using $\mathbb{Z}^M$ lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



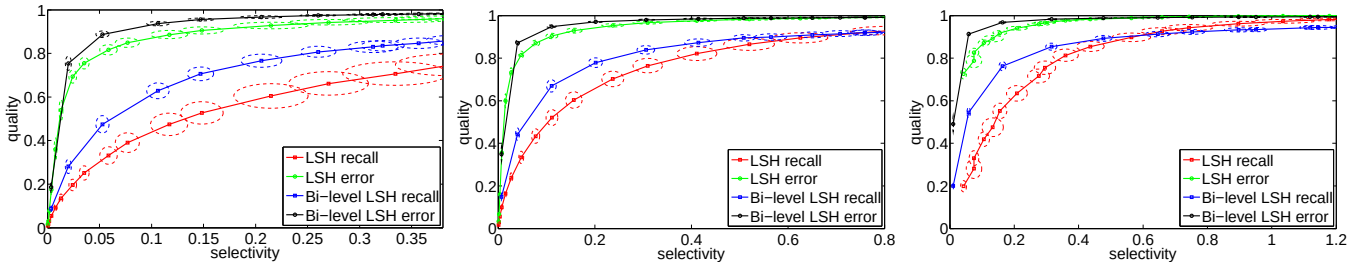
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 9: Quality comparison between our multiprobed Bi-level LSH and standard multiprobed LSH when using E_8 lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



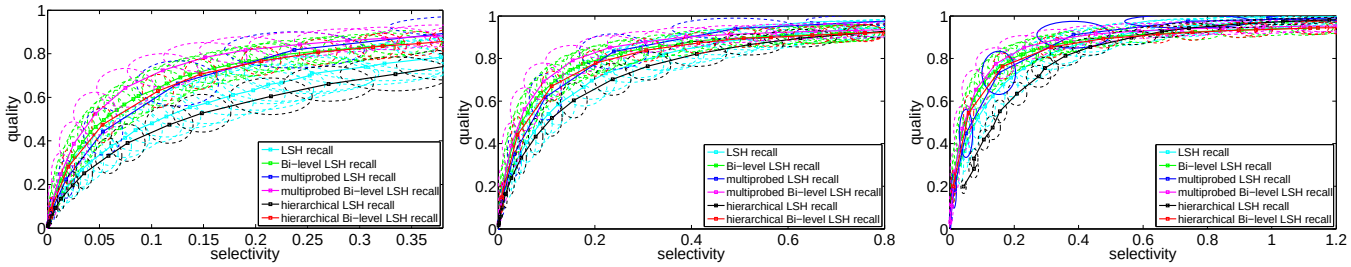
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 10: Quality comparison between our hierarchical Bi-level LSH and standard hierarchical LSH when using $\mathbb{Z}^M$ lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



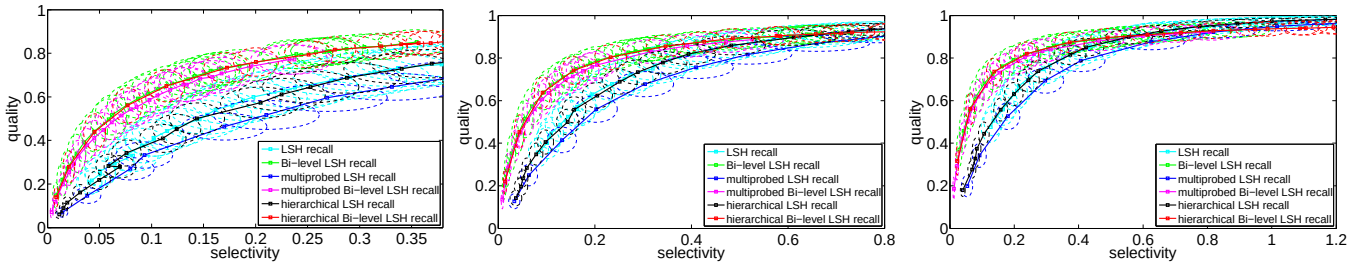
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 11: Quality comparison between our hierarchical Bi-level LSH and standard hierarchical LSH when using E_8 lattice. $M = 8$ and Bi-level LSH partitions the dataset into 16 groups in the first level. Each figure shows the selectivity–recall curves and the selectivity–error curves for both methods. The ellipses show the standard deviations of selectivity and recall/error ratio caused by random projections.



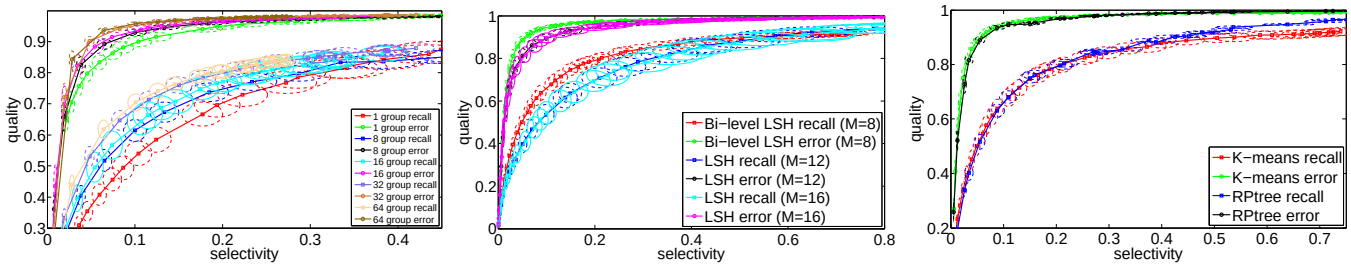
(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 12: Variance caused by queries when using $\mathbb{Z}^M$ lattice. We compare six methods: standard LSH, multiprobed LSH, standard LSH + Morton hierarchy, Bi-level LSH, multiprobed Bi-level LSH, Bi-level LSH + Morton Hierarchy.



(a) $L = 10$ (b) $L = 20$ (c) $L = 30$

Figure 13: Variance caused by queries when using E_8 lattice. We compare six methods: standard LSH, multiprobed LSH, standard LSH + E_8 hierarchy, Bi-level LSH, multiprobed Bi-level LSH, Bi-level LSH + E_8 Hierarchy.



(a) (b) (c)

Figure 14: all $L = 20$ (a) different group number (b) different M (c) RP-tree and K-means

our algorithm on more real-world datasets, including images, textures, videos, etc. We also need to design efficient out-of-core algorithms to handle very large datasets (e.g. $>$ 100GB) on one PC.

8. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Symposium on Foundations of Computer Science*, pages 459–468, 2006.

[2] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *International Conference on World Wide Web*, pages 651–660, 2005.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *International Conference on Database Theory*, pages 217–235, 1999.

[4] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *International Conference on Machine Learning*, pages 97–104, 2006.

[5] M. Connon and P. Kumar. Fast construction of k-nearest neighbor graphs for point clouds. 16:599–608, 2010.

[6] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Symposium on Theory of Computing*, pages 537–546, 2008.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[8] A. Dhesi and P. Kar. Random projection trees revisted. In *Advances in Neural Information Processing Systems*, 2010.

[9] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *Conference on Information and Knowledge Management*, pages 669–678, 2008.

[10] O. Egecioglu and B. Kalantari. Approximating the diameter of a set of points in the euclidean space. *Information Processing Letters*, 32:205–211, 1989.

[11] Y. Freund, S. Dasgupta, M. Kabra, and N. Verma. Learning the structure of manifolds using random projections. In *Advances in Neural Information Processing Systems*, 2007.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Data Bases*, pages 518–529, 1999.

[13] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing. In *International Conference on Knowledge Discovery and Data Mining*, pages 1129–1138, 2010.

[14] H. Jégou, L. Amsaleg, C. Schmid, and P. Gros. Query adaptative locality sensitive hashing. In *International Conference on Acoustics, Speech and Signal Processing*, pages 825–828, 2008.

[15] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *International Conference on Multimedia*, pages 209–218, 2008.

[16] Z. Kang, W. T. OOi, and Q. Sun. Hierarchial, non-uniform locality sensitive hashing and its application to video identification. In *International Conference on Multimedia and Expo*, pages 743–746, 2004.

[17] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *International Conference on Management of Data*, pages 369–380, 1997.

[18] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Symposium on Theory of Computing*, pages 599–608, 1997.

[19] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. In *Advances in Neural Information Processing Systems*, 2009.

[20] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *International Conference on Computer Vision*, pages 2130–2137, 2009.

[21] S. Liao, M. A. Lopez, and S. T. Leutenegger. High dimensional similarity search with space filling curves. In *International Conference on Data Engineering*, pages 615–622, 2001.

[22] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *International Conference on Very Large Data Bases*, pages 950–961, 2007.

[23] J. Pan and D. Manocha. Gpu-based bi-level lsh for high-dimensional nearest neighbor computation. Technical report, UNC Chapel Hill, May 2011.

[24] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31:1348–1358, 2010.

[25] M. Raginsky and S. Lazebnik. Locality-senstive binary codes from shift-invariant kernels. In *Advances in Neural Information Processing Systems*, 2009.

[26] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems*, 2007.

[27] R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50:969–978, 2009.

[28] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transations on Database Systems*, 35:20:1–20:46, 2010.

[29] S. S. Vempala. *The random projection method*. American Mathematical Society, 2004.

[30] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *International Conference on Very Large Data Bases*, pages 194–205, 1998.

[31] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Advances in Neural Information Processing Systems*, 2008.

[32] D. Yan, L. Huang, and M. I. Jordan. Fast approximate spectral clustering. In *International Conference on Knowledge Discovery and Data Mining*, pages 907–916, 2009.