Scalability and Total Recall with Fast CoveringLSH *

Ninh Pham
IT University of Copenhagen
Denmark
ndap@itu.dk

Rasmus Pagh
IT University of Copenhagen
Denmark
pagh@itu.dk

ABSTRACT

Locality-sensitive hashing (LSH) has emerged as the dominant algorithmic technique for similarity search with strong performance guarantees in high-dimensional spaces. A drawback of traditional LSH schemes is that they may have *false negatives*, i.e., the recall is less than 100%. This limits the applicability of LSH in settings requiring precise performance guarantees. Building on the recent theoretical "CoveringLSH" construction that eliminates false negatives, we propose a fast and practical covering LSH scheme for Hamming space called *Fast CoveringLSH (fcLSH)*. Inheriting the design benefits of CoveringLSH our method avoids false negatives and always reports all near neighbors. Compared to CoveringLSH we achieve an asymptotic improvement to the hash function computation time from $\mathcal{O}(dL)$ to $\mathcal{O}(d + L \log L)$, where d is the dimensionality of data and L is the number of hash tables. Our experiments on synthetic and real-world data sets demonstrate that *fcLSH* is comparable (and often superior) to traditional hashing-based approaches for search radius up to 20 in high-dimensional Hamming space.

1. INTRODUCTION

Similarity search is a fundamental ingredient in algorithms for a wide range of computer applications, including machine learning, database management, information retrieval, and pattern recognition and analysis. This problem has become increasingly important and challenging in the era of big data since the use of computational resources such as storage and power becomes critical. Content-based image retrieval systems now have to answer similarity queries over billion-size image databases [25]. Large-scale collaborative filtering engines have to deal with tens of millions users' data [5]. The emergence of big data adds to both research and commercial applications the challenges of *scale* and *accuracy* for efficient similarity search.

*The research leading to these results has received funding from the European Research Council under the EU 7th Framework Programme, ERC grant agreement no. 614331.

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983742

In most such applications data can be represented or approximated as high-dimensional binary vectors, and Hamming distance is used as a similarity measure. For instance, a near-duplicate detection system uses hashing techniques [4, 11] to represent documents as binary vectors, and identifies them as near-duplicates if their Hamming distances are smaller than a threshold radius. In content-based image retrieval systems, a standard approach is to learn short binary codes to represent image objects such that the Hamming distance between codes reflects their neighborhood or semantic similarity in the original space, see e.g. [19] and its references. Retrieving similar images can be efficiently done by simply returning all images with codes within a small Hamming distance of the code of the query image.

Similarity search in Hamming space dates back to Minsky and Papert [13], who referred to it as the *approximate dictionary* problem. The generalization to arbitrary spaces is now known as *near neighbor search*. Due to the "curse of dimensionality", the performance of indexing techniques based on data or space partitioning generally degrades as dimensionality increases, and is eventually no better than a simple linear search [23].

Since 1998 *locality-sensitive hashing* (LSH) [9] has emerged as a basic primitive for near neighbor search in high-dimensional space. It alleviates the effects of the "curse of dimensionality" by considering an *approximate* variant, and obtains sub-linear time for the approximation problem. Since its first introduction, many LSH-based methods for near neighbor search have been proposed. We refer to [6, 20] for an up-to-date overview of developments. However, a drawback of classical LSH-based methods is the probabilistic guarantees that result in false negatives (i.e., the recall is below 100%). This limits applicability of LSH in settings requiring high accuracy or precise performance guarantees, e.g., fingerprint recognition, entity resolution, and plagiarism detection.

Although the requirement of perfect recall ratio has not often been the primary focus when studying similarity search in Hamming space, there are many applications where this setting is relevant. For the problem of large-scale image search and recognition, learning binary codes for images to preserve their neighborhood or semantic similarity (see [14] and its references) is widely used due to the simplicity of the representation and fast query processing. False negative findings in querying a binary code can degrade the performance of classification and retrieval tasks. In fact, such methods often perform brute-force search for answering near neighbor queries. Recently researchers have found that the binary codes must be long enough (hundreds of bits) to preserve dis-

crimination power and to achieve good performance, see [24] and its references. As such applications arise in large-scale image data sets, the problem of scaling up similarity search in high-dimensional Hamming space is getting more important and more challenging.

In a recent theoretical study, *CoveringLSH* [17] was proposed to address the issue of false negatives in LSH for Hamming space. Instead of independently selecting bit positions from high-dimensional binary vectors as the classic LSH method [9], CoveringLSH carefully chooses correlated bit positions that "cover" all possible positions of r differences, and thus eliminate false negatives. An issue of CoveringLSH is that it requires the computation of L hash values of d bits, where d is the dimensionality of data and L is the number of hash tables. This becomes a bottleneck for large dimensions, since evaluation time proportional to dL is unavoidable.

Fast CoveringLSH. This paper presents *Fast CoveringLSH* (fcLSH), a fast and practical evolution of CoveringLSH that scales much better to high dimensions. Inheriting the design benefits of CoveringLSH, fcLSH can not only answer *approximate* near neighbor search with provable sub-linear guarantees, but also report the *exact* set of all near neighbors. In addition, for low dimensions where $d \leq L$, fcLSH achieves higher precision than CoveringLSH. Our experiments on synthetic and real-world data sets demonstrate that fcLSH is comparable and often superior to traditional LSH approaches for search radius up to 20 in high-dimensional Hamming space.

Technical contributions. Observe that for $d \gg \log(n)$ we can decrease the size of the hash values from d to $\mathcal{O}(\log n)$ bits each, while not significantly changing collision probabilities, by applying universal hashing [3]. In order to avoid intermediate results of dL bits we show how to *interweave* a carefully chosen universal hash function with the Fast Hadamard Transform, such that L hash values of $\mathcal{O}(\log n)$ bits are computed directly. Since the Hadamard matrix is related to the projection family used by CoveringLSH, the values computed in this way are identical to those obtained by hashing the d-bit hash values to $\mathcal{O}(\log n)$ bits. This approach achieves an asymptotic improvement to hash function computation time from $\mathcal{O}(dL)$ to $\mathcal{O}(d + L \log L)$, for d dimensions and L hash tables.

2. BACKGROUND AND PRELIMINARIES

2.1 Problem Setting

We study the problem of near neighbor search in Hamming space under Hamming distance. Due to the "curse of dimensionality", many proposed solutions for *exact* near neighbor search in high-dimensional space become slower than simple linear search. In order to trade precision for speed, approximate versions of near neighbor search have been widely investigated in the literature, and locality-sensitive hashing-based methods have emerged as the most widely used solutions for such problems. The first approximate version, called c-approximate r-near neighbor search, is defined as follows.

DEFINITION 1. *(c-approximate r-near neighbor or (c, r)-NN) Given a set $S \subset \{0,1\}^d$, $|S| = n$, the Hamming distance function d, and parameters $r > 0$, $c > 1$, $\delta > 0$, construct a data structure such that, given any query $\boldsymbol{q} \in \{0,1\}^d$, if there exists a point $\boldsymbol{x} \in S$ and $d(\boldsymbol{x}, \boldsymbol{q}) \leq r$, it reports some point $\boldsymbol{y} \in S$ where $d(\boldsymbol{y}, \boldsymbol{q}) \leq cr$ with probability $1 - \delta$.*

We note that (c, r)-NN problem has two approximation factors, consisting of the approximation of distance by a factor of c, and the approximation of the result set determined by the success probability $1 - \delta$. Due to the approximation of distance, this problem formulation may give undesirable quality of results. By setting $c = 1$, the second approximate version, called r-near neighbor reporting, has more practical applications [19, 21] since it reports *all* points within distance r to the query.

DEFINITION 2. *(r-near neighbor reporting or r-NN) Given a set $S \subset \{0,1\}^d$, $|S| = n$, the Hamming distance function d, and parameters $r > 0$, $\delta > 0$, construct a data structure that, given any query $\boldsymbol{q} \in \{0,1\}^d$, return each point $\boldsymbol{x} \in S$ where $d(\boldsymbol{x}, \boldsymbol{q}) \leq r$ with probability $1 - \delta$.*

We call this the "exact" r-NN problem in case $\delta = 0$, otherwise it is the "inexact" r-NN to distinguish with the approximation term of (c, r)-NN problem. Note that the inexact factor here is due to reporting *each* near neighbors, determined by the success probability $1 - \delta$.

This work investigates the possibility of an *exact* guarantee for r-NN problem in order to report every point $\boldsymbol{x} \in S$ where $d(\boldsymbol{x}, \boldsymbol{q}) \leq r$. It is worth noting that solving the exact r-NN problem implies an exact solution to the nearest neighbor problem with comparable performance by building several solutions for different radii [1]. For convenience of notation, we are now using a bold letter for a binary vector (e.g., $\boldsymbol{v}$) to distinguish it from a scalar quantity (e.g., v). In what follows, we talk about *near* points at distance at most r (those that should be reported), *c-near* points at distance between r and cr, an *far* points at distance larger than cr.

2.2 Locality-sensitive Hashing Functions

LSH is one of the most widely used approaches to near neighbor search in high-dimensional space because it is able to break the $\mathcal{O}(n)$ barrier for the (c, r)-NN problem.

DEFINITION 3. (Indyk and Motwani [9]) *Fix a distance function $d : \mathbb{U} \times \mathbb{U} \to \mathbf{R}$. For positive reals r, c, p_1, p_2, and $p_1 > p_2$, $c > 1$, a family of functions $\mathcal{H}$ is (r, cr, p_1, p_2)-*sensitive *if for uniformly chosen $h \in \mathcal{H}$ and all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{U}$:*

- *If $d(\boldsymbol{x}, \boldsymbol{y}) \leq r$ then $\mathbf{Pr}[h(\boldsymbol{x}) = h(\boldsymbol{y})] \geq p_1$;*
- *If $d(\boldsymbol{x}, \boldsymbol{y}) \geq cr$ then $\mathbf{Pr}[h(\boldsymbol{x}) = h(\boldsymbol{y})] \leq p_2$.*

The classic LSH family for Hamming distance uses a bit sampling approach [7, 9]. It is simply the family of all the projections of points to one dimension, i.e., a hash function value is just a random bit sample. That is, given a point $\boldsymbol{x} = \{x_1, \ldots, x_d\}$, the bit sampling LSH family $\mathcal{B}$ with parameters $p_1 = 1 - r/d$, $p_2 = 1 - cr/d$ is constructed as:

$$\mathcal{B} = \left\{ h : \{0,1\}^d \to \{0,1\} \,|\, h(\boldsymbol{x}) = x_i \text{ for some } i \in \{1, \ldots, d\} \right\}.$$

The performance of LSH-based algorithms is governed by the parameter $\rho = \log p_1 / \log p_2$, and constructing an LSH family with small ρ automatically leads to the improved algorithms for the (c, r)-NN problem. For the bit sampling family $\mathcal{B}$, $\rho \approx 1/c$ which is optimal for data-independent LSH in Hamming space [16].

The classical LSH-based algorithm for near neighbor search problem is as follows. Given an LSH family $\mathcal{H}$, construct L hash tables by hashing data points using L hash functions g_j, $j = 1, \ldots, L$, by setting $g_j = \left(h_j^1, \ldots, h_j^k\right)$, where h_j^i, $i = 1, \ldots, k$, are chosen randomly from the LSH family $\mathcal{H}$. To

process a query $\boldsymbol{q}$, one needs to retrieve candidate points from the bucket $g_j(\boldsymbol{q})$ in the jth hash table, $j = 1, \ldots, L$. For the candidate set retrieved, a filtering procedure is performed to remove false positives. There are different filtering strategies corresponding to (c, r)-NN and r-NN problems [1].

Strategy 1: Stop searching after finding the first $3L$ points (including duplicates) and return the point with minimum distance to the query $\boldsymbol{q}$.

Strategy 2: For each distinct point $\boldsymbol{x}$ from the candidate set, compute $d(\boldsymbol{x}, \boldsymbol{q})$ and report $\boldsymbol{x}$ if $d(\boldsymbol{x}, \boldsymbol{q}) \leq r$.

Strategy 1 that interrupts the search after retrieving $3L$ points (including duplicates) is of significant importance in theory because it introduces a sub-linear time algorithm with suitable choices of k and L for the (c, r)-NN problem [7, 9]. In particular, it runs in $\mathcal{O}(n^{\rho})$ time where $\rho = \log p_1 / \log p_2$ if we suitably choose $k = \mathcal{O}(\log n)$, $L = \mathcal{O}(n^{\rho})$.

Strategy 2 enables us to solve the r-NN problem, which has more practical applications. It provides better result quality since all reported points are within distance r to the query point. It might run in $\mathcal{O}(n)$ time in the worst case, but for many natural data sets, proper settings of k and L still result in a sub-linear query time [1]. However, Strategy 2 can introduce false negatives if some near points do not collide with the query under any hash function. That limits the use of LSH in applications requiring high accuracy or precise performance guarantees.

For practical implementation[1], the value $k = \mathcal{O}(\log n)$ is large in a typical setting. One can reduce the time of checking collision and the amount of memory for bucket identification from $\mathcal{O}(k)$ to $\mathcal{O}(1)$ by using an associated universal hash function to hash a k-bit hash value into an integer. Moreover, since the domain of the hash function g_j is too large to store all possible buckets explicitly, and we only need to store non-empty buckets, we use a hash table to contain these non-empty buckets. Given a prime P and random integers b_i, $i = 1, \ldots, k$, from the interval $\{0, \ldots, P-1\}$, we use hash functions of the form:

$$p(x_1, \ldots, x_k) = b_1 \cdot x_1 + \cdots + b_k \cdot x_k \mod P. \tag{1}$$

According to [3], this family is universal which means that the probability of collision is small if P is sufficiently large.

2.3 CoveringLSH

In very recent work [17], a novel LSH scheme was proposed to solve the exact r-NN problem. This method always introduces a collision for every pair of binary vectors within a given radius r. Instead of independently selecting bit positions as in the bit sampling approach, CoveringLSH carefully chooses correlated bit positions so that it can "cover" *all* possible positions of r differences, which implies an *exact* guarantee for the r-NN problem when used with Strategy 2.

DEFINITION 4. *An LSH family $\mathcal{A}$ is* r-covering *if for every two binary vectors $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^d$ with Hamming distance $d(\boldsymbol{x}, \boldsymbol{y}) \leq r$, there exists $g \in \mathcal{A}$ such that $g(\boldsymbol{x}) = g(\boldsymbol{y})$.*

The proposed scheme relies on a *random* mapping $m : [d] \to \{0,1\}^{r+1}$ that maps bit positions to binary vectors of length $r+1$. This r-covering LSH family, $\mathcal{A}$, consists of $2^{r+1} - 1$ correlated hash functions via the mapping m. Each hash function is associated with a binary vector of length d, denoted by $\boldsymbol{g}_v$, $v = 1, \ldots, 2^{r+1} - 1$ of the form

$$\boldsymbol{g}_v = (\langle m(1), \boldsymbol{v} \rangle, \langle m(2), \boldsymbol{v} \rangle, \cdots, \langle m(d), \boldsymbol{v} \rangle), \tag{2}$$

where $\langle m(i), \boldsymbol{v} \rangle = \sum_{j=1}^{r+1} m(i)_j v_j \mod 2$ is the dot product modulo 2 of two vectors $m(i)$ and $\boldsymbol{v}$. The hash value of a given binary vector $\boldsymbol{x}$ is simply the binary vector produced by the bit-wise **AND** operation, i.e., $g_v(\boldsymbol{x}) = \boldsymbol{g}_v \wedge \boldsymbol{x}$. The $2^{r+1} - 1$ hash functions of $\mathcal{A}$ correspond to all distinct non-zero binary vectors $\boldsymbol{v} \in \{0,1\}^{r+1} \setminus \{\boldsymbol{0}\}$ or equivalently binary representations of $v \in \{1, \ldots, 2^{r+1} - 1\}$. Hence, the non-zero binary vector $\boldsymbol{v}$ or the corresponding integer v is used to index the vth hash function $\boldsymbol{g}_v$, and we will use them interchangeably.

EXAMPLE 2.1. *Given the two binary vectors $\boldsymbol{x} = 0011$ and $\boldsymbol{q} = 1010$, we have that $d(\boldsymbol{x}, \boldsymbol{q}) = 2$. A 2-covering LSH family uses a random mapping $m : [4] \to \{0,1\}^3$, e.g., $m(1) = 011, m(2) = 100, m(3) = 101, m(4) = 001$, to construct 7 hash functions as follows:*

$$\boldsymbol{g}_1 = (\langle m(1), 001 \rangle, \langle m(2), 001 \rangle, \langle m(3), 001 \rangle, \langle m(4), 001 \rangle) = 1011,$$
$$\boldsymbol{g}_2 = (\langle m(1), 010 \rangle, \langle m(2), 010 \rangle, \langle m(3), 010 \rangle, \langle m(4), 010 \rangle) = 1000,$$
$$\boldsymbol{g}_3 = 0011, \boldsymbol{g}_4 = 0110, \boldsymbol{g}_5 = 1101, \boldsymbol{g}_6 = 1110, \boldsymbol{g}_7 = 0101.$$

There is one collision between $\boldsymbol{x}$ and $\boldsymbol{q}$ corresponding to $\boldsymbol{g}_4 = 0110$ since $\boldsymbol{g}_4 \wedge \boldsymbol{x} = \boldsymbol{g}_4 \wedge \boldsymbol{q} = 0010$. Note that the 2-covering LSH family can cover all possible positions of 2 differences in 4-dimensional Hamming space.

THEOREM 1. [17, Lemma 3.2] *For every mapping $m : [d] \to \{0,1\}^{r+1}$, the family $\mathcal{A}$ built as above is r-covering.*

The new r-covering LSH scheme can not only eliminate the problem of false negatives but also essentially *match* the complexity bound of the seminal LSH construction of Indyk and Motwani [9] if $cr = \log n$. This is due to Theorem 2.

THEOREM 2. [17, Theorem 3.1] *For any two binary vectors $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^d$ and a random mapping $m : [d] \to \{0,1\}^{r+1}$, $\mathcal{A}$ has two following properties:*

1. *If $d(\boldsymbol{x}, \boldsymbol{y}) \leq r$ then $\mathbf{Pr}[\exists g \in \mathcal{A} : g(\boldsymbol{x}) = g(\boldsymbol{y})] = 1$.*
2. *$\mathbf{E}[|\{g \in \mathcal{A} \mid g(\boldsymbol{x}) = g(\boldsymbol{y})\}|] < 2^{r+1-d(\boldsymbol{x}, \boldsymbol{y})}$.*

It is obvious that, for the setting where $cr = \log n$, the number of hash functions is $2^{r+1} - 1 \approx 2n^{1/c}$ and the total expected number of collisions for the far points among all hash functions is at most $2^r \approx n^{1/c}$. This implies an efficient sub-linear algorithm for solving the (c, r)-NN problem with constant success probability, like the classic LSH schemes. In addition, the r-covering LSH scheme with Strategy 2 will answer the r-NN problem with an *exact* guarantee, returning *all* points within distance r to the query.

The basic r-covering scheme needs time $\mathcal{O}(d)$ to construct one hash function (see Equation (2)). In practice, the dimensionality of binary data can be high, e.g., documents, recommendation data sets. Also, an embedding process to Hamming space can require high dimensionality, e.g., embedding ℓ_1-norm into Hamming space by a *unary* representation [7], semantic hashing to embed images into Hamming space [8, 24]. This issue demands significant computational resources for computing r-covering hash codes.

[1] E2LSH. http://www.mit.edu/∼andoni/LSH/.

2.4 Hadamard Codes

The Hadamard code is an error-correcting code that enables efficient and reliable message transmission over noisy channels. Here we aim at using Hadamard codes to construct LSH hash functions, and we will not use its error-correcting properties. Instead, we explain how to generate Hadamard codes over the binary alphabet $\{0,1\}$ and how to leverage it to construct CoveringLSH hash functions.

Given a binary vector $\boldsymbol{v} \in \{0,1\}^{r+1}$, the Hadamard code maps $\boldsymbol{v}$ into a binary vector $\text{Had}(\boldsymbol{v})$ of length 2^{r+1} using an encoding function $\text{Had} : \{0,1\}^{r+1} \to \{0,1\}^{2^{r+1}}$. In particular, $\text{Had}(\boldsymbol{v})$ is generated as follows:

$$\text{Had}(\boldsymbol{v}) = \left(\langle a(0), \boldsymbol{v}\rangle, \langle a(1), \boldsymbol{v}\rangle, \cdots, \langle a(2^{r+1}-1), \boldsymbol{v}\rangle\right), \quad (3)$$

where $a(i)$, $i = 0, \ldots, 2^{r+1}-1$, are *all* possible binary vectors in $\{0,1\}^{r+1}$, and $\langle a(i), \boldsymbol{v}\rangle$ is the dot product modulo 2 of two vectors $a(i)$ and $\boldsymbol{v}$.

Consider the hash function vector $\boldsymbol{g}_v$ in Equation (2) and the Hadamard code $\text{Had}(\boldsymbol{v})$ in Equation (3). It is observed in [17] that $\boldsymbol{g}_v$ can be seen as sampling a subset of $\text{Had}(\boldsymbol{v})$ since the random mapping m is a subset of $\{0,1\}^{r+1}$. We note that the Hadamard code of a binary vector $\boldsymbol{v}$ corresponds to the vth row of the so-called *Hadamard matrix* $\mathbf{H}$ of the same size using the mapping $1 \mapsto -1$ and $0 \mapsto 1$. Conversely, we can use the Hadamard matrix of size $2^{r+1} \times 2^{r+1}$ with the reverse mapping as Hadamard codes for vectors $\boldsymbol{v} \in \{0,1\}^{r+1}$. The next section will exploit this relation and show how to use Hadamard codes and the fast Hadamard transform `FHT()` to efficiently construct r-covering LSH families.

3. ALGORITHM

A typical case. Let us now present an example of a typical setting of image search where Hadamard codes can be used as r-covering LSH functions *without* any modifications. Suppose that we have a set of binary vectors $S \subseteq \{0,1\}^8$. Given a query $\boldsymbol{q}$, we would like to find all points within distance $r = 2$ from $\boldsymbol{q}$. We generate Hadamard codes $\mathbf{C}_{7,8}$ by using the rows of Hadamard matrix $\mathbf{H}_{8,8}$ as described above, and remove the first row to avoid trivial collisions. We see that $\mathbf{C}_{7,8}$ is a 2-covering LSH family. (We *will* in fact use the first column of the Hadamard matrix to simplify the fcLSH description and construction; in the practical implementation we later discard it due to its trivial collision.)

$$\mathbf{C}_{7,8} = \begin{pmatrix} 0&1&0&1&0&1&0&1 \\ 0&0&1&1&0&0&1&1 \\ 0&1&1&0&0&1&1&0 \\ 0&0&0&0&1&1&1&1 \\ 0&1&0&1&1&0&1&0 \\ 0&0&1&1&1&1&0&0 \\ 0&1&1&0&1&0&0&1 \end{pmatrix}$$

It is obvious that there exists at least one collision for every pair of vectors within distance 2. This implies an algorithm for the 2-NN problem without false negatives using the $\mathbf{C}_{7,8}$ LSH family (see Example 3.1). Note that, in this case, the mapping $m(i)$ of the r-covering LSH scheme is simply the vector representing i in binary.

EXAMPLE 3.1. *Given the two binary vectors* $\boldsymbol{x} = 0011\underline{0}0\underline{1}1$, $\boldsymbol{y} = 0011\underline{0}00\underline{1}$ *and the query vector* $\boldsymbol{q} = 00111010$, *we have that* $d(\boldsymbol{x}, \boldsymbol{q}) = 2$, $d(\boldsymbol{y}, \boldsymbol{q}) = 3$. *Given the 2-covering LSH family* $\mathbf{C}_{7,8}$, *there is one collision between* $\boldsymbol{x}$ *and* $\boldsymbol{q}$ *(i.e.,* $g_3(\boldsymbol{x}) = g_3(\boldsymbol{q}) = 00100010$*) corresponding to the 3rd row of* $\mathbf{C}_{7,8}$, *and there is no collision between* $\boldsymbol{y}$ *and* $\boldsymbol{q}$.

In a typical setting for large-scale image search, suppose that we have a set S of $n = 2^{24}$ vectors from $\{0,1\}^{128}$. Given a query $\boldsymbol{q}$, we may wish to search all vectors in S within distance $r = 6$ from $\boldsymbol{q}$. Since an exhaustive search in Hamming balls with $r = 6$ would take much more time than just linear search, we settle for a 4-approximate similarity search ($cr = \log n$).

The 6-covering LSH requires 127 hash functions and we use a random column-based permutation of the Hadamard codes $\mathbf{C}_{127,128}$ as the LSH family. Theorem 2 shows that near vectors within radius 6 always collide with $\boldsymbol{q}$ in at least 1 hash function. Moreover, in expectation, a far-away vector at distance larger than 24 has collision probability at most $1/2^{24} = 1/n$ under each hash function. This means that the r-covering LSH scheme can be used for efficiently answering the *exact* r-NN search by pruning almost all far vectors.

The general case. We now consider the general case of r-covering LSH schemes for answering exact r-NN queries. Note that the constraint $cr = \log n$ affects the efficiency of r-covering LSH-based algorithms because it determines the pruning power. Moreover, it is also the key factor governing the "best tradeoff" between space and time complexity for the (c, r)-NN problem. Another hurdle for r-covering LSH schemes is that high dimensionality d requires significant hash function computation time.

We use a method from [17] to handle the constraint $cr = \log n$. When $cr < \log n$, we increase the radius r by replicating each bit of both data and query points $\lfloor \log n/cr \rfloor$ times. On the other hand, when $cr > \log n$, we leverage the pigeonhole principle by first permuting and then partitioning the dimensions of both data and query points into $\lceil cr/\log n \rceil$ parts (see Example 3.2). Then we independently build LSH data structures for each partition and the candidate vectors are generated for each partition.

EXAMPLE 3.2. *Given a binary vector* $\boldsymbol{q} = 0011$, *replicating* $\boldsymbol{q}$ *2 times returns a new vector* $\boldsymbol{q}^{(2)} = 00110011$. *A random permutation of* $\boldsymbol{q}$ *gives* $\boldsymbol{q}' = 0110$. *Partitioning* $\boldsymbol{q}'$ *into 2 parts returns two vectors* $\boldsymbol{q}_1 = 01, \boldsymbol{q}_2 = 10$.

After replicating or partitioning the dimensions, we use a new query radius r' where $cr' \approx \log n$. Denote by d' the new dimensionality of data and query points. If $d' > 2^{r'+1}$, we will need a random mapping $m : [d'] \to [2^{r'+1}]$ that randomly samples d' columns from the Hadamard codes $\mathbf{C}_{2^{r'+1}-1,2^{r'+1}}$ to form the r'-covering LSH family. On the other hand, if $d' \leq 2^{r'+1}$, we can leverage a 0-padding trick to increase the dimensionality to $2^{r'+1}$ without changing r', and simply use the Hadamard codes $\mathbf{C}_{2^{r'+1}-1,2^{r'+1}}$ with columns randomly permuted as the r'-covering LSH family, as in the typical case above. In both cases, if d' is large, it affects the hashing cost, i.e., computing the hash value and identifying the bucket corresponding to the query. The trick of converting long binary hash values into integers, see the Equation (1), still requires $\mathcal{O}\left(d'2^{r'+1}\right)$ time. To address this problem, we propose to use the fast Hadamard transform for quickly computing integer hash values in $\mathcal{O}\left(d' + r'2^{r'+1}\right)$ time, which is asymptotically faster when $d' > r'$.

Algorithm 1 Pre-processing algorithm

Require: A vector $\boldsymbol{q} = \{q_1, \ldots, q_d\}$, radius $r > 0$, approximation ratio $c > 1$, and data set size n

1: **if** $cr < \log n$ **then**
2: $\quad \boldsymbol{q}$ is replicated $t = \lfloor \log(n)/cr \rfloor$ times to form a new vector $\boldsymbol{q}^{(t)} = \underbrace{\boldsymbol{q} \ldots \boldsymbol{q}}_{t \text{ times}}$
3: **else if** $cr > \log n$ **then**
4: $\quad$ Randomly permute $\boldsymbol{q}$
5: $\quad \boldsymbol{q}$ is partitioned into $t = \lceil cr/\log n \rceil$ parts to form t new vectors:
$\quad \boldsymbol{q}_1 = \{q_1, \ldots, q_{\lfloor d/t \rfloor}\}, \ldots, \boldsymbol{q}_t = \{q_{(t-1)\lfloor d/t \rfloor}, \ldots, q_d\}$
6: **end if**

3.1 Construction

As elaborated above, we need to satisfy the constraint $cr = \log n$ in order to achieve high pruning power like the classic LSH scheme. We handle this issue by simply replicating or partitioning the dimensionality of both data and query points to increase or decrease the radius r to be approximately $\log(n)/c$, as illustrated in Algorithm 1.

For simplicity of notation, let us denote by d and r the new dimensionality of data and the new query radius, respectively, after pre-processing data to satisfy $cr \approx \log n$. We now present two variants of the fcLSH scheme: a general construction using a random mapping $m : [d] \to [2^{r+1}]$ for $d > 2^{r+1}$ as introduced in [17] and a specific construction using a random permutation $m : [2^{r+1}] \to [2^{r+1}]$ for $d \leq 2^{r+1}$. In both cases, we exploit the fast Hadamard transform for fast computation of hash functions.

The general construction for $d > 2^{r+1}$. Recall that the basic r-covering LSH family requires $L = 2^{r+1} - 1$ hash functions and the construction of a hash function g_v relies on a random mapping $m : [d] \to \{0,1\}^{r+1}$ and dot products modulo 2 between $m(i)$ and $\boldsymbol{v}$, described in Equation (2). This procedure is identical to randomly sampling d positions among 2^{r+1} positions from $\text{Had}(\boldsymbol{v})$, the Hadamard code of the vector $\boldsymbol{v}$. This implies that we can use a new random *mapping* $m : [d] \to [2^{r+1}]$ and rely on a simple construction without computing d dot products as follows.

$$\boldsymbol{g}_v = \{\text{Had}(\boldsymbol{v})_{m(1)}, \text{Had}(\boldsymbol{v})_{m(2)}, \ldots, \text{Had}(\boldsymbol{v})_{m(d)}\}. \quad (4)$$

The specific construction for $d \leq 2^{r+1}$. It is obvious that any collision caused by the random mapping m yields more collisions for both close points and far points. That might slightly degrade the performance of filtering mechanisms. In typical settings of content-based image retrieval applications where $d \leq 2^{r+1}$, we can combine the 0-padding trick with a random *permutation* $m : [2^{r+1}] \to [2^{r+1}]$ over columns of the Hadamard codes $\mathbf{C}_{2^{r+1}-1,2^{r+1}}$ to achieve better results than the construction in Equation (4). This idea is illustrated in Algorithm 2 (lines 7–8).

Fast computation of hash functions. We use a conventional hash function to map a binary hash value of length d into an integer hash value in order to reduce the amount of memory for bucket identification and time complexity of searching a bucket in a hash table. A naïve approach requires $\mathcal{O}(dL)$ time complexity, see Equation (1). We show that we can reduce this cost to $\mathcal{O}(d + L \log L)$ by using the fast Hadamard transform `FHT()`, as illustrated in Algorithm 2.

We generate a random seed vector $\boldsymbol{b}$ to convert binary hash codes into integers (line 1) and compute a new vector $\widetilde{\boldsymbol{q}} = \boldsymbol{q} * \boldsymbol{b}$ by component-wise multiplication, i.e., $(\boldsymbol{q} * \boldsymbol{b})_i = q_i b_i$. Non-zero entries of $\widetilde{\boldsymbol{q}}$ correspond to 1s in $\boldsymbol{q}$. If the dimensionality of data is greater than the length of Hadamard codes, i.e., 2^{r+1}, we evaluate the random mapping m on each dimension i of $\widetilde{\boldsymbol{q}}$, and sum up colliding entries to form the new sketch vector $\boldsymbol{t}$ of length 2^{r+1} (line 5). Otherwise, we apply 0-padding trick on $\widetilde{\boldsymbol{q}}$ and randomly permute it to get $\boldsymbol{t}$ (line 8). We note that applying a random permutation on $\boldsymbol{q}$ is equivalent to applying a random permutation on the Hadamard codes, because we are only concerned with collisions. `FHT`($\boldsymbol{t}$) is then used to reduce the cost of computing 2^{r+1} integer hash values (line 10). Finally, we ignore the first element corresponding to the first row of the Hadamard matrix to get $L = 2^{r+1} - 1$ integer hash values (line 11).

Algorithm 2 Generating hash values using the fast Hadamard transform `FHT()`

Require: A point $\boldsymbol{q} \in \{0,1\}^d$, a prime P
Ensure: A vector $\boldsymbol{h}$ of $L = 2^{r+1} - 1$ integer hash values

1: Pick a random integer-valued vector $\boldsymbol{b} \in [P]^d$
2: Compute a new integer-valued vector $\widetilde{\boldsymbol{q}} = \boldsymbol{q} * \boldsymbol{b}$ by component-wise multiplication
3: **if** $d > 2^{r+1}$ **then**
4: $\quad$ Pick a random mapping $m : [d] \to [2^{r+1}]$
5: $\quad$ Compute a sketch vector $\boldsymbol{t}$ where $t_j = \sum_{i:m(i)=j} \widetilde{\boldsymbol{q}}_i$
6: **else**
7: $\quad$ Pick a random permutation $m : [2^{r+1}] \to [2^{r+1}]$
8: $\quad \boldsymbol{t} := m(\widetilde{\boldsymbol{q}}\ ; \mathbf{0}^{2^{r+1}-d})$
9: **end if**
10: $\boldsymbol{h} = \frac{1}{2}\left(\|\widetilde{\boldsymbol{q}}\|_1 \mathbf{1} - \texttt{FHT}(\boldsymbol{t})\right) \mod P$
11: Remove the first element from $\boldsymbol{h}$

Time complexity analysis. Denote by $nnz(\boldsymbol{q})$ the number of non-zero entries of vector $\boldsymbol{q}$. The running time at line 11 using the fast Hadamard transform `FHT()` is $\mathcal{O}(L \log L)$. The other computational costs are bounded by $\mathcal{O}(nnz(\boldsymbol{q}))$. The total running time is $\mathcal{O}(nnz(\boldsymbol{q}) + L \log L)$, which can be compared to $\mathcal{O}(nnz(\boldsymbol{q})L)$ of the basic r-covering LSH scheme [17]. When $nnz(\boldsymbol{q}) > \log L$, fcLSH is sufficiently faster than the basic r-covering scheme.

3.2 Theoretical Analysis

Now we sketch a theoretical analysis of the correctness of fcLSH. We first show that the general and the specific construction are efficient r-covering LSH schemes with the two properties stated in Theorem 2. Note that the first property guarantees that fcLSH always eliminates false negatives and reports all near neighbors for the r-NN problem. The second property says that fcLSH has the same pruning power as the classic LSH scheme [9]. Then we argue that Algorithm 2 computes exactly the same results as using the universal hash function $p()$ defined in Equation (1) to convert a binary hash value into an integer. A more detailed proof and discussion can be seen in the full version [18].

LEMMA 1. *Both general and specific constructions described above give r-covering LSH families. These families satisfy properties 1 and 2 in Theorem 2.*

PROOF. Using the proofs of [17, Lemma 3.2] and [17, Theorem 3.1], we can verify the claim. □

It is worth noting that in a typical setting where $d = 2^{r+1}$, the size of $\mathbf{C}_{2^{r+1}-1,2^{r+1}}$ is close to the smallest possible for an

r-covering LSH family. Observe that we have $\binom{2^{r+1}}{r}$ possible sets of r differences, and each row of $\mathbf{C}_{2^{r+1}-1,2^{r+1}}$ can cover at most $\binom{2^r}{r}$ such sets. This means that the number of hash functions needed is at least $\binom{2^{r+1}}{r}/\binom{2^r}{r} > 2^r$, which is within a factor of 2 from the upper bound. This implies that the specific construction often gives better results than the general construction.

Next, we argue that Algorithm 2 produces r-covering LSH families. Before presenting lemmas, let us describe the main technical insight used in Algorithm 2. Consider the ideal case where $d = 2^{r+1}$, and recall that the Hadamard code matrix $\mathbf{C}$ can be generated by the Hadamard matrix $\mathbf{H}$ with the same size by mapping $1 \mapsto 0$ and $-1 \mapsto 1$. If we let $\mathbf{1}$ denote the matrix with 1 in every entry, we have $\mathbf{C} = (\mathbf{1} - \mathbf{H})/2$. Given any binary vector $\boldsymbol{q}$, the hash value of $\boldsymbol{q}$ under the hash function vector $\mathbf{C}_v$ (vth row of $\mathbf{C}$) is $g_v(\boldsymbol{q}) = \mathbf{C}_v \wedge \boldsymbol{q}$. Using the universal hash function $p()$ in Equation (1), we need a prime P and a random seed vector $\boldsymbol{b}$ for computing $\boldsymbol{b} \cdot g_v(\boldsymbol{q}) \mod P$. This means that we need to compute the matrix-vector multiplication $\mathbf{C}\widetilde{\boldsymbol{q}}$, where $\widetilde{\boldsymbol{q}} = \boldsymbol{q} * \boldsymbol{b}$ is a component-wise product, as follows:

$$\mathbf{C}\widetilde{\boldsymbol{q}} = \frac{1}{2}(\mathbf{1} - \mathbf{H})\widetilde{\boldsymbol{q}} = \frac{1}{2}\left\|\widetilde{\boldsymbol{q}}\right\|_1 \mathbf{1} - \frac{1}{2}\mathbf{H}\widetilde{\boldsymbol{q}}. \tag{5}$$

LEMMA 2. *Given a prime P and any random seed vector $\boldsymbol{b} \in [P]^d$, Algorithm 2 computes the same hash values as using $p()$ in Equation (1) on the r-covering LSH scheme introduced in [17].*

PROOF. Since the random permutation used in the specific construction is a special case of the random mapping used in the general construction, we need only prove the claim for the general construction.

Given any binary vector $\boldsymbol{q}$, we let $\widetilde{\boldsymbol{q}} = \boldsymbol{q} * \boldsymbol{b}$. It is clear that the contribution of $b_i q_i$ to the integer hash value is determined by the random mapping value $m(i)$. We form a vector $\boldsymbol{\xi}_i \in \mathbf{N}^{2^{r+1}}$ corresponding to the contribution of $b_i q_i$ whose entry at position $m(i)$ is $q_i b_i$ and the others are zero. The integer-value hash values of $\boldsymbol{q}$ is then computed as follows:

$$\begin{aligned}\mathbf{C}\left(\sum_{i=1}^{d}\boldsymbol{\xi}_i\right) &= \frac{1}{2}\sum_{i=1}^{d}(b_i q_i - \mathbf{H}\boldsymbol{\xi}_i) = \frac{1}{2}\left(\sum_{i=1}^{d} b_i q_i - \sum_{i=1}^{d}\mathbf{H}\boldsymbol{\xi}_i\right)\\ &= \frac{1}{2}\sum_{i=1}^{d} b_i q_i - \frac{1}{2}\mathbf{H}\sum_{i=1}^{d}\boldsymbol{\xi}_i = \frac{1}{2}\left\|\widetilde{\boldsymbol{q}}\right\|_1 \mathbf{1} - \frac{1}{2}\mathbf{H}\boldsymbol{t},\end{aligned}$$

where the vector $\boldsymbol{t}$ is computed by $t_j = \sum_{i:m(i)=j} b_i q_i$. Applying `FHT()` on the second term proves the claim. □

COROLLARY 1. *Given a sufficiently large prime P, a construction of fcLSH provided by Algorithm 2 is an r-covering LSH scheme with properties 1 and 2 of Theorem 2.*

4. EXPERIMENT

We implemented fcLSH in C++ and conducted experiments on an Intel Xeon Processor E5-1650 v3 with 64GB of RAM. We compared the performance of hashing-based algorithms for reporting all near neighbors, including our fcLSH scheme, the basic r-covering LSH [17], the classic LSH scheme [9], and the multi-index hashing approach [15] on synthetic and real-world data sets. Each result is the average of 5 runs over a query set of an algorithm.

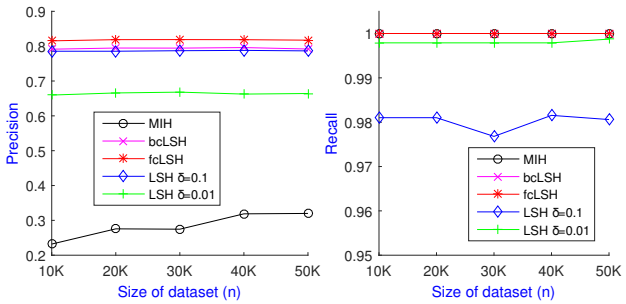

Figure 1: Comparison of precision/recall rate between fcLSH and bcLSH without pre-processing, MIH, the classic LSH with $\delta = 0.1$ and $\delta = 0.01$ on synthetic data sets of $n = 10K - 50K$ and $r = 6$.

Table 1: Hash function computation time

Method	fcLSH	bcLSH	LSH	MIH
Time	$\mathcal{O}(d + L\log L)$	$\mathcal{O}(dL)$	$\mathcal{O}(kL)$	$\mathcal{O}(d)$

4.1 Experiment Setup

We consider alternative hashing-based approaches with performance guarantees in Hamming space for comparison. The following algorithms are used.

- **fcLSH**: Our method with fast computation of hash function using `FHT()`.
- **bcLSH**: The basic covering construction [17] based on random samples from Hadamard codes.
- **LSH**: The classic LSH [9] using bit sampling approach.
- **MIH**: The recent multi-index hashing approach [15] running in sub-linear time for exact r-NN over uniformly distributed data sets.

Note that MIH is an alternative to exhaustive search in Hamming balls over data sub-dimensions. Based on the pigeonhole principle, MIH partitions data dimensions to reduce the radius, which is similar to our approach. However, the sub-linear guarantees of MIH is based on the strong assumption of uniform distribution of data points which is not true in many natural data sets [10, 12].

Parameter settings. It is obvious that each hashing-based method achieves the best performance given the proper choices of parameters. Since such proper choices primarily depend on the distance distribution between queries and data points, we use suggested settings as below.

- For the CoveringLSH schemes, including fcLSH and bcLSH [17], we only need the partition trick when r is large (say, $r \geq 10$) since we might not have enough space for $L = 2^{r+1} - 1$ hash tables.
- For classic LSH, we simply set the number of hash tables $L = 2^{r+1} - 1$ for the sake of comparison. The number of bit samples is set as $k = \left\lceil \log(1 - \delta^{1/L}) / \log(1 - r/d) \right\rceil$ where δ is the false negative ratio[2].
- For MIH, the number of partitions is $\lceil d/\log_2 n \rceil$ as suggested in [15].

Cost measurement. To report all near neighbors, we follow the Strategy 2. In general, for each query, any hashing-based approach needs to process the following operations:

[2]http://www.mit.edu/~andoni/LSH/manual.pdf

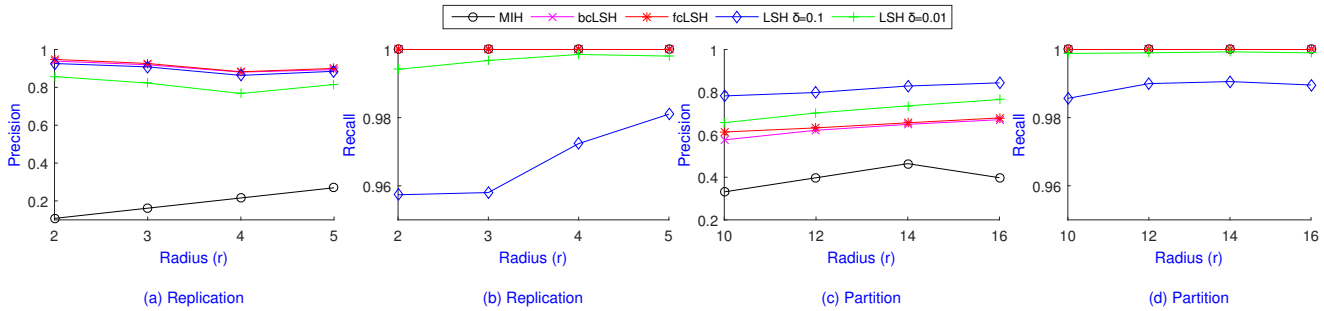

Figure 2: Comparison of precision/recall rate between fcLSH and bcLSH with pre-processing, MIH, classic LSH with $\delta = 0.1$ and $\delta = 0.01$ on the synthetic data set of $n = 64K$ points.

Table 2: Data set properties

Data sets	n	d	Binarization
ANN_SIFT1M	1M	128	LSH
Webspam	0.35M	254	LSH
Enron	$\sim$ 40K	$\sim$ 28K	Word freq. > 10
MovieLens	$\sim$ 0.23M	$\sim$ 140K	Rating > 2

- **Step S1:** Compute hash functions to identify the bucket of the query on each of the L hash tables.
- **Step S2:** Look up in each hash table the points in the bucket of the query, and merge them together for duplicate elimination to form a list of candidates.
- **Step S3:** Compute the actual distance between candidates and the query to report near neighbor points.

We decompose the total search cost per query into 3 cost components of the three main steps above. The cost of S1 is dependent on the dimensionality of data and the parameter settings for each algorithm which can be analyzed precisely (see Table 1), whereas the costs of S2 and S3 significantly depend on the data distribution and the distance distribution between query and data points, respectively. Since the data sets used in our experiment are both in low-dimensional and high-dimensional space, we focus on the cost of S2 and S3.

The cost of S2, called C_{lookup}, is for merging and removing duplicates since very close points might collide many times in different hash tables. Typically, we use a bitmap string of n bits to remove such duplicates [15, 21]. Every time a candidate is found, we set the bit corresponding to that candidate. Thus this cost is proportional to the number of collisions `#Collisions` over all hash tables.

The cost of S3, called C_{check}, is proportional to the number of *distinct* candidates `#Candidates` returned from step S2. Dependent on the dimensionality d, the size of candidates, and cache and disk access implementation, this cost may or may not dominate C_{lookup}. Hence, for the sake of comparison, we report separately these two main costs for each algorithm. More detailed experimental evaluations can be seen in the full version [18].

4.2 Data Sets

We evaluated the performance of fcLSH using synthetic data sets and 4 real-world data sets from images, text, and recommendation systems. Properties of these data sets are summarized in Table 2. See [18] for more details.

- **Synthetic** contains uniformly distributed binary data sets of dimension 128.
- **ANN_SIFT1M [10]** contains 1 million 128-dimensional SIFT feature vectors of images.
- **Webspam**[3] contains 350,000 web documents.
- **Enron**[4] contains a collection of about 0.4 million emails.
- **MovieLens**[5] contains ratings applied to 140,214 movies by 234,834 users.

We randomly remove 50 points from the ANN_SIFT1M and Webspam data sets, and 100 points from the Enron and MovieLens data sets to use them as query points in our performance study. We need more query points for the latter cases for the sake of comparison, since with small radius (up to 20), there are some query points that do not have any near neighbors. The ground truth for each query point is computed by a linear scan of the entire data sets.

4.3 Synthetic Data Sets

We carried out experiments to evaluate the accuracy and efficiency of our constructions with and without a pre-processing step (replicating/partitioning), over synthetic data sets for the task of reporting all near neighbors. We used precision/recall rates to measure the performance of hashing-based methods, including fcLSH, bcLSH, MIH, and classic LSH with recall ratio of 90% (i.e., $\delta = 0.1$) and 99% (i.e., $\delta = 0.01$), for a wide range of query radii and data set sizes. We note that if we ignore the C_{lookup} cost, the precision ratio corresponds to the speedup compared to linear search.

Figure 1 displays the precision/recall rate of algorithms for reporting points within distance $r = 6$ from a query. The number of hash tables for the LSH-based method is $L = 2^{r+1} - 1 = 127$ whereas that of MIH is at most 10. It is obvious that LSH-based approaches achieve almost 3 times higher precision than the MIH approach. In other words, C_{check} of MIH is around 3 times larger than LSH-based approaches. In addition, fcLSH achieves slightly better precision than both bcLSH and classic LSH. Classic LSH shows a tradeoff between precision and recall rate where the one with recall ratio 99% has lower precision than that of recall ratio 90%. Regarding recall ratio, both CoveringLSH schemes and MIH achieve perfect recall whereas classic LSH obtains a high recall ratio (at least 97.5%) but not 100%.

Figure 2 shows the precision/recall rate of fcLSH and bcLSH with preprocessing tricks (replication and partition) and other algorithms. We replicated {4, 3, 2, 2} times corresponding to $r = 2, 3, 4, 5$, respectively. This leads to space

[3]http://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets
[4]http://archive.ics.uci.edu/ml/data sets/Bag+of+Words
[5]http://grouplens.org/data sets/movielens/

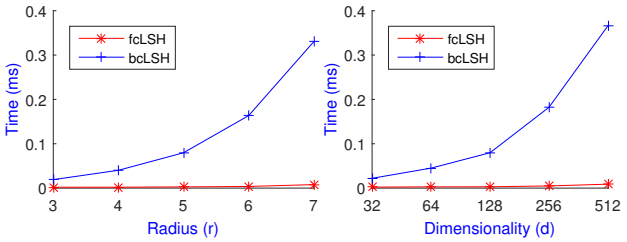

Figure 3: Comparison of hash function computation time per query between fcLSH and bcLSH on synthetic data sets of $n = 64K$: $d = 128$ and $r = 3 - 7$ (left), and $r = 5$ and $d = 32 - 512$ (right).

overhead $L = 511, 1023, 511, 2047$ for LSH-based methods compared to $L = 8$ of MIH, and explains why the precision ratio of MIH fluctuates. The results are very similar to the case without a pre-processing step: LSH schemes show their superiority compared to MIH, fcLSH has slightly higher precision than bcLSH, and the classic LSH approach always introduces false negatives. We used 2 partitions for $r = 10, 12, 14, 16$. The number of hash tables is $L = 126, 254, 510, 1022$ for LSH-based methods, and $L = 8$ for MIH. Again, LSH-based approaches outperform MIH regarding to precision ratio. However, the precision of r-covering approaches is worse than classic LSH approaches since the partition trick introduces more unexpected collisions. This difference is at most $n^{\ln(4)}$ in the worst-case data sets as analyzed in [17].

Figure 3 concludes the experimental results on synthetic data sets by showing the hash function computation time per query between two approaches: fcLSH and bcLSH. It is clear that fcLSH gives substantially faster hash function computation time due to the fast Hadamard transform for a wide range of d and r.

4.4 Real-world Data Sets

The experiments on synthetic data sets illustrate that fcLSH achieves better performance than bcLSH: less hash function computation time and higher precision with total recall. Hence, we now use fcLSH as the representative of r-covering LSH to compare to other approaches on real-world data sets. Since the recall ratios of classic LSH with $\delta = 0.1$ and with $\delta = 0.01$ are almost the same and very high, we only use the classic LSH with $\delta = 0.1$ for comparison.

We observe that the replication trick often results in more collisions since it uses more hash tables. In practice, the pruning power of LSH-based approaches is primarily dependent on the distance distribution between data points and query points. Moreover, the space usage for indexes is usually limited by RAM. This requires the query radius r to be rather small (say, up to 10) for large data sets (up to 1M points). Therefore, we do not usually need the pre-processing step for small r and only use the partition trick for large r.

As discussed in Subsection 4.1, we used the total number of collisions and the distinct candidate set size, denoted by `#Collisions` and `#Candidates`, respectively, to measure separately the two main costs C_{lookup} and C_{check}. Due to memory constraints we only consider search radius up to 20 on all data sets. We use 1 partition (without pre-processing step) for $r < 10$ with $L = 2^{r+1} - 1$, and 2 partitions for $r \geq 10$ with $L = 2(2^{\lfloor r/2 \rfloor + 1} - 1)$ for LSH-based methods.

Table 3: Recall ratios on ANN_SIFT1M 64 bits

Radius	5	6	7	8	9
fcLSH / MIH	1	1	1	1	1
Classic LSH	0.96	0.94	0.93	0.93	0.92

For MIH, we used the standard setting, i.e., $L = \lceil d/\log_2 n \rceil$ hash tables.

4.4.1 *Low-dimensional data sets*

This subsection compares the performance of 4 approaches: covering LSHs (fcLSH and bcLSH), classic LSH with $\delta = 0.1$, and MIH on the ANN_SIFT1M (images) and Webspam (texts) data sets. Since we aim at measuring the efficiency of these algorithms in low-dimensional space, we generated binary data sets of {64, 128} bits for ANN_SIFT1M, and {256, 512} bits for Webspam. Due to similar results on both data sets, we only report representative recall ratios of ANN_SIFT1M 64 bits for small radii $r \in \{5, \ldots, 9\}$, as shown in Table 3. The results confirm that classic LSH cannot avoid false negatives while the other approaches do.

Figure 4 shows the two main costs per query on the ANN_SIFT1M and Webspam data sets with different dimensionality. Since fcLSH and bcLSH have the same hash values, the two main costs are identical. It is obvious that LSH-based approaches outperform the MIH approach on the ANN_SIFT1M data set. For the 64-bit version, `#Candidates` and `#Collisions` for MIH are dramatically larger than for fcLSH and classic LSH. In particular, MIH's `#Collisions` is up to around 7 times larger than that of LSH-based approaches. The largest gap starts at $r = 9$. This is because MIH uses 3 hash tables and $r \geq 9$ indicates a new radius $r' = 3$ for each partition. This change influences both `#Candidates` and `#Collisions` of MIH. As the theoretical analysis shows that r-covering LSH schemes and classic LSH have similar pruning power for far points, their performance, including `#Candidates` and `#Collisions`, are very similar for $r = 5 - 10$. For 2 partitions, coveringLSH is slightly worse than classic LSH due to the probability of splitting distances unevenly over the partitions. However, when we target to approach 100% recall ratio, covering LSH schemes clearly outperform MIH, obtaining up to 7 and 14 times speedup regarding `#Collisions` and `#Candidates`, respectively.

On the Webspam data set, `#Candidates` returned by MIH is orders of magnitude larger than for the LSH-based approaches. This is because d is rather large, so the estimated cost of MIH, $\mathcal{O}((d/r)^r)$, tends to be very large, even comparable to the data set size. Hence, in terms of guaranteeing perfect recall, covering LSH provides superior performance compared to MIH. Compared to classic LSH, the performance of covering LSH is similar when using 1 partition and slightly worse with 2 partitions. In particular, `#Candidates` and `#Collisions` provided by fcLSH using 2 partitions is approximately twice that of classic LSH.

Figure 5 shows the superiority of LSH-based methods (fcLSH, bcLSH, classic LSH with $\delta = 0.1$) to the MIH method with respect to the average CPU time per query in milliseconds on the ANN_SIFT1M data set. It is obvious that LSH-based approaches run at least 2 times faster than MIH for radii $r = 10 - 15$. On the 64-bit version, since the cost C_{check} and C_{lookup} of LSH-based approaches are very similar, fcLSH provides superior performance compared to

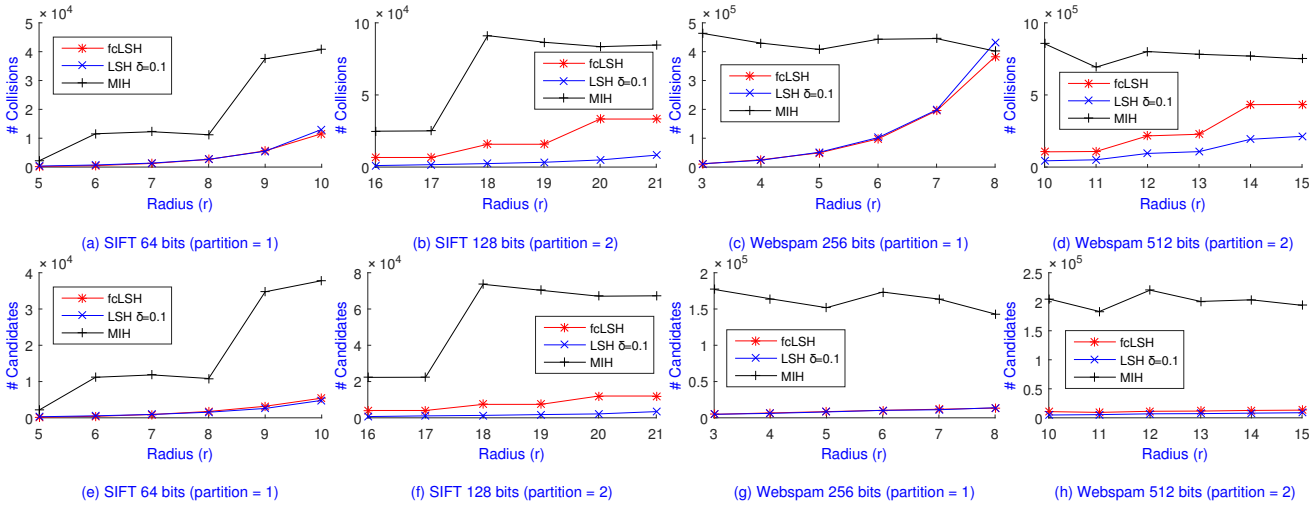

Figure 4: Comparison of the number of collisions and distinct candidate set size for fcLSH, MIH, classic LSH with $\delta = 0.1$ on two data sets: ANN_SIFT1M and Webspam.

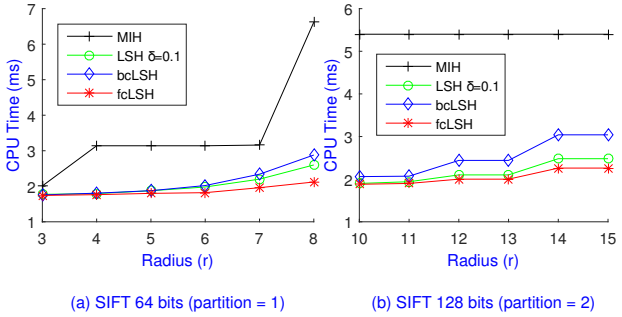

Figure 5: Comparison of CPU Time (ms) per query of 4 approaches: fcLSH, bcLSH, classic LSH with $\delta = 0.1$, and MIH on ANN_SIFT1M data set.

bcLSH and classic LSH due to the fast hash computation. For $r = 6 - 8$, bcLSH is slightly slower than classic LSH. This is because the increase in the of number of hash tables, $L = 2^{r+1} - 1$ leads to a slightly larger gap in hash computation time, dL of bcLSH compared to kL of classic LSH. On the 128-bit version, classic LSH is favorably compared to bcLSH because `#Candidates` and `#Collisions` provided by bcLSH considerably increase due to partitioning. However, fcLSH still gains substantial advantages from the fast hash computation and outperforms bcLSH and classic LSH.

4.4.2 *High-dimensional data sets*

This subsection studies the ability of scale and accuracy of 3 approaches, fcLSH, bcLSH and classic LSH with $\delta = 0.1$ on two high-dimensional binary data sets: Enron email and MovieLens. Since the data sets are very high-dimensional, the MIH approach is outperformed by the simple linear search and we do not report the results for MIH here. Due to similar results on the two data sets, we report representative recall ratios of MovieLens for small radii $r = 2 - 7$, as shown in Table 4. The results once again confirm that fcLSH always eliminates false negatives while classic LSH cannot.

Since the data sets are very high-dimensional and distance

Table 4: Recall ratios on Movielens

Radius	3	4	5	6	7
fcLSH	1	1	1	1	1
Classic LSH	0.97	0.99	0.99	0.98	0.98

computation is time consuming, the cost C_{check} dominates the cost C_{lookup}. Hence we only report `#Candidates` and CPU Time (ms) per query of the 3 approaches, as shown in Figure 6. This result again supports our theoretical comparison of fcLSH to classic LSH. `#Candidates` of covering LSH is slightly smaller than classic LSH in the case of 1 partition on MovieLens but up to 2.5 times larger when using 2 partitions on Enron. However, fcLSH still provides superior performance compared to bcLSH and classic LSH due to the fast hash computation process. It is also clear that classic LSH outperforms bcLSH due to less hash computation time. In general, fcLSH is favorably compared to classic LSH but is superior to both MIH and bcLSH in settings requiring precise performance guarantees.

5. RELATED WORK

Due to the "curse of dimensionality", one typically uses linear search for (exact) near neighbor search in high-dimensional Hamming space [14, 19, 22]. To trade precision for speed, *approximate* retrieval is widely investigated in the research literature, and LSH [9] is a widely used technique due to its attractive "tradeoff" between time and space. However, false negatives findings limit the applicability of LSH in settings requiring precise performance guarantees.

Exact search in Hamming space has recently attracted research attention since there is growing interest in learning binary codes for large-scale image search and recognition. Recently, Norouzi et al. [15] proposed the MIH approach which partitions each data vector to reduce the search radius, and then applies exhaustive search. Although MIH has sub-linear running time behaviour for *uniformly* distributed data sets, it does not work well in general. This is because its performance relies on the ability to select a small number

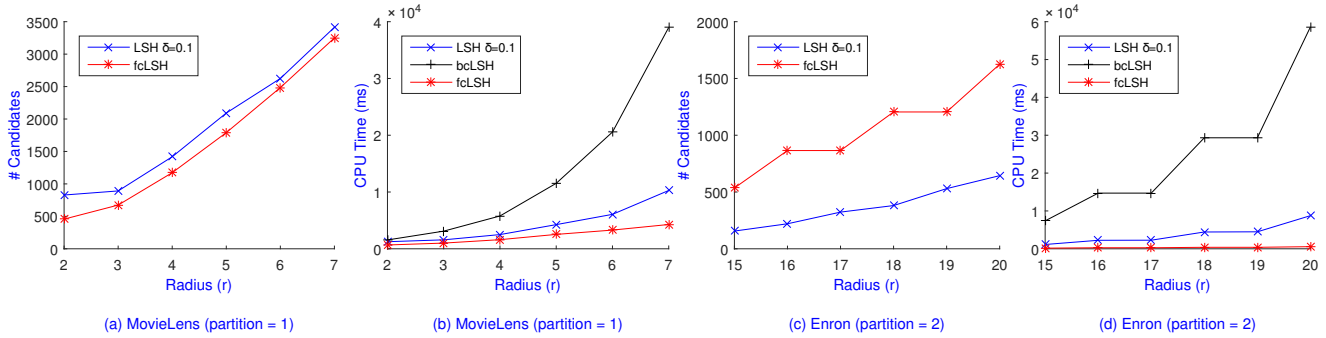

Figure 6: Comparison of the distinct candidate set size and CPU time (ms) between fcLSH and classic LSH with $\delta = 0.1$ on two data sets: Enron emails and Movielens.

of random bit positions (around $\log n$) for which there are almost no collisions between the query point and points in the data set – an assumption that is not true in general.

Arasu et al. [2] proposed the idea that randomly permuting the dimensions of data vectors increases the robustness of partitioning, and make performance guarantees possible for data sets that are not uniformly distributed. They combined this idea with another level of partitioning within which a "brute force" r-cover is found. Recently, Pagh [17] proposed CoveringLSH, as described in Section 2.3, which is always better than the method of Arasu et al. when $r > 2$. This is because the number of hash values needed by [2] is $\binom{2r}{r} \approx 4^r/\sqrt{r}$, which is much larger than 2^{r+1} required by r-covering scheme. In the case of several partitions, Arasu et al. leave it unspecified how to best choose the parameters of their method, so it is really a family of methods. For these reasons we have not implemented this method.

6. CONCLUSIONS

This paper proposes *Fast CoveringLSH*, a fast and practical LSH scheme for Hamming space. Inheriting the design benefits from CoveringLSH, our method avoids false negatives and always reports *all* near neighbors. Our main technical contribution is asymptotic improvement to the hash function computation time from $\mathcal{O}(dL)$ to $\mathcal{O}(d + L \log L)$, for d dimensions and L hash tables. Our experiments on synthetic and real-world data sets demonstrate the efficiency of fcLSH in comparison with traditional hashing-based approaches for search radius up to 20 in high-dimensional Hamming space.

An obvious open direction is to extend our work to other spaces or similarity measures, aiming at rigorous performance guarantees without false negatives. Since the recent covering LSH framework demands a large number of hash tables for large radii, another interesting question would be to reduce the space usage to linear (or near-linear) in the data size while maintaining the property of total recall.

7. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 2008.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[3] L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *STOC*, 1977.

[4] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[5] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In *WWW*, 2007.

[6] J. Gao, H. V. Jagadish, B. C. Ooi, and S. Wang. Selective hashing: Closing the gap between radius search and k-nn search. In *KDD*, 2015.

[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[8] Y. Gong, S. Kumar, H. A. Rowley, and S. Lazebnik. Learning binary codes for high-dimensional data using bilinear projections. In *CVPR*, 2013.

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[10] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 2011.

[11] P. Li and C. König. b-bit minwise hashing. In *WWW*, 2010.

[12] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[13] M. Minsky and S. Papert. *Perceptrons - an introduction to computational geometry*. MIT Press, 1987.

[14] M. Norouzi, D. J. Fleet, and R. Salakhutdinov. Hamming distance metric learning. In *NIPS*, 2012.

[15] M. Norouzi, A. Punjani, and D. J. Fleet. Fast exact search in Hamming space with multi-index hashing. *TPAMI*, 2014.

[16] R. O'Donnell, Y. Wu, and Y. Zhou. Optimal lower bounds for locality-sensitive hashing (except when q is tiny). *TOCT*, 6(1):5, 2014.

[17] R. Pagh. Locality-sensitive hashing without false negatives. In *SODA*, 2016.

[18] N. Pham and R. Pagh. Scalability and total recall with fast coveringLSH. Full version, arXiv:1602.02620, 2016.

[19] R. Salakhutdinov and G. Hinton. Semantic hashing. *Int. J. Approx. Reasoning*, 2009.

[20] A. Shrivastava and P. Li. Asymmetric LSH for sublinear time maximum inner product search. In *NIPS*, 2014.

[21] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *PVLDB*, 2013.

[22] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *CVPR*, 2008.

[23] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[24] F. X. Yu, S. Kumar, Y. Gong, and S. Chang. Circulant binary embedding. In *ICML*, 2014.

[25] L. Zhang and Y. Rui. Image search—from thousands to billions in 20 years. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2013.