PUFFINN: Parameterless and Universally Fast FInding of Nearest Neighbors

Martin Aumüller
IT University of Copenhagen, Denmark
maau@itu.dk

Tobias Christiani
IT University of Copenhagen, Denmark
tobc@itu.dk

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

Michael Vesterli
IT University of Copenhagen, Denmark
miev@itu.dk

Abstract

We present PUFFINN, a parameterless LSH-based index for solving the k-nearest neighbor problem with probabilistic guarantees. By parameterless we mean that the user is only required to specify the amount of memory the index is supposed to use and the result quality that should be achieved. The index combines several heuristic ideas known in the literature. By small adaptions to the query algorithm, we make heuristics rigorous. We perform experiments on real-world and synthetic inputs to evaluate implementation choices and show that the implementation satisfies the quality guarantees while being competitive with other state-of-the-art approaches to nearest neighbor search. We describe a novel synthetic data set that is difficult to solve for almost all existing nearest neighbor search approaches, and for which PUFFINN significantly outperform previous methods.

2012 ACM Subject Classification Mathematics of computing → Probabilistic algorithms; Theory of computation → Design and analysis of algorithms; Theory of computation → Nearest neighbor algorithms

Keywords and phrases Nearest Neighbor Search, Locality-Sensitive Hashing, Adaptive Similarity Search

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.10

Related Version `https://arxiv.org/pdf/1906.12211.pdf`

Supplement Material `https://github.com/puffinn/esa-paper`

Funding The research leading to these results has received funding from the European Research Council under the European Union's 7th Framework Programme (FP7/2007-2013) / ERC grant agreement no. 614331. BARC, Basic Algorithms Research Copenhagen, is supported by the VILLUM Foundation grant 16582.
Rasmus Pagh: Supported by Investigator Grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation.

27th Annual European Symposium on Algorithms (ESA 2019).
Editors: Michael A. Bender, Ola Svensson, and Grzegorz Herman; Article No. 10; pp. 10:1–10:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Our results

The k-nearest neighbor (k-NN) problem has been an object of intense research, both from theoretical and applied computer scientists. There exist many implementations of k-NN data structures that perform very well in specific scenarios (see the related work section), but all implementations that we are aware of suffer from one or more of the following drawbacks:

- Not scalable to large, high-dimensional data sets.
- Not runtime-robust in the sense that query time may degrade to that of a linear search even for input distributions that are known to allow search in sublinear time.
- Not recall-robust in the sense that there are input distributions that obtain low (less than 50%) recall.
- Performance bounds only hold for well-chosen values of certain parameters that depend on the data set as well as the query distribution.

PUFFINN combines several insights from recent theoretical research on k-NN data structures into a data structure that addresses these drawbacks. Our contributions are as follows:

1. we present a parameterless and universal locality-sensitive hashing-based (LSH) implementation that solves the k-NN problem with probabilistic guarantees (Section 3)
2. we prove the correctness of an adaptive query mechanism building on top of the LSH forest data structure described by Bawa et al. in [5] (Section 3)
3. we describe an adaptive filtering approach to decrease the number of expensive distance computations (Section 4)
4. we propose a difficult dataset for the 1-NN problem that exposes weaknesses in known heuristics (Section 5)
5. we provide a detailed experimental study of our approach, evaluating design choices and relating it to the performance of other state-of-the-art approaches to k-NN search (Section 5)

Prior to this work, only subsets of these ideas have been implemented. Our main contribution on the theoretical side is that we make certain heuristics, such as the query algorithm described in [5], rigorous. While some ideas of adaptive query algorithms have been discussed before [18], they made assumptions on the data and query distribution; our methods work for k-NN search in its full generality. On the practical side, we shed light on the empirical performance of theoretical ideas with regard to possible speed-ups of an LSH implementation. Our final implementation is parameterless in the sense that it only requires the user to specify the space available for the data structure and the required quality guarantee. Our implementation is competitive to state-of-the-art k-NN algorithms; in particular, it is as performant as previous LSH solutions that are not parameterless and do not provide guarantees on the quality of the result.

1.2 Related work

There exist many fundamentally different approaches to nearest neighbor search. Popular techniques range from approximate tree-based methods, such as kd-trees [6] and M-trees [14] with an early stopping criterion [33], to random projection trees [16], to graph-based approaches [24, 21], and finally hashing-based approaches, for example using LSH [20]. Each paradigm comes with performant implementations and we will introduce some of them in

Section 5 when we are evaluating our implementation. Since the focus of the present paper is designing a provably correct LSH implementation we will focus on existing methods in this realm. See the benchmarking paper by Aumüller et al. [4] for a more detailed overview over other approaches and their performance on real-world datasets.

Locality-sensitive hashing. LSH was introduced by Indyk and Motwani in [20]. We sketch the basic idea here. An LSH data structure consists of several independent repetitions of space partitions using LSH functions. An LSH function maps a data point to a hash code such that closer points are more likely to collide than far away points. In the LSH framwork, $K \geq 1$ locality-sensitive hash functions are concatenated to increase the gap between the collision probability of "close" points and "far away" points. For solving the (c, r)-near neighbor problem [20], a certain concatenation length K is fixed according to the number of points in the data set, the approximation factor c, and the strength of the hash family at hand. From the value K and the hash family one can compute how many repetitions L (using independent hash functions) have to be made to guarantee that a close point is found with constant probability. The theoretical literature on LSH has mostly focused on solving the approximate near neighbor problem which can be be used to solve the approximate *nearest* neighbor problem through a reduction [19]. In this paper we use LSH to solve the exact k-nearest neighbor problem with probabilistic guarantees.

LSH implementations. Implementations of LSH evolved over the years with new advances in the theory of LSH. One of the first popular implementations, dubbed E2LSH [2], was tailored for Euclidean space and included automatic parameter tuning of the K parameter based on subsampling the dataset. From this K parameter, other parameter such as the number of repetitions are derived. It solves the problem of reporting all points within a distance r (specified during preprocessing) from the query and uses potentially large space on large datasets [2]. The multiprobing approach introduced by Dong et al. in [18] allowed for implementations in which the space parameter can be fixed as in our approach. Their parameter tuning relies on the assumption that there is a certain distance distribution between queries and data points. LSHkit was the first implementation using this idea [17]. A new LSH family for angular distance on the unit sphere motivated the development of the FALCONN library [1]. It contains highly optimized routines for efficient hash function computation, and supports multiprobing.

None of these approaches give guarantees on the query procedure if the query set is different with respect to distance distributions from the one seen during index building. Our data structure can be seen as a modified version of the LSH forest introduced by Bawa et al. in [5]. Instead of using a single hash length K, an individual repetition is a trie built on the hash codes of the data points. Our query algorithm replaces the heuristic candidate collection of a predefined size with a rigorous termination criterion.

The cost of evaluating an LSH function differs widely. It ranges from $O(1)$ time for the bitsampling approach in Hamming space [20], over $O(d)$ for random hyperplane hashing [10], to $O(d^2)$ for cross-polytope LSH [1]. For the latter, the authors of [1] proposed a heuristic version that decreases the running time to $O(d \log d)$. Another approach to reduce hashing time is to build a hashing oracle that returns the necessary KL hash function values necessary to query the LSH data structure, but builds those from a smaller set of independent hash functions. Christiani [12] described two approaches that reduce the amount of independent hash functions needed to produce these hash values from KL to $K\sqrt{L}$ (using tensoring as in [3]) or $O(\log^2 n)$ (using the pooling approach in [15]). While the E2LSH framework uses a variant of tensoring, we are not aware of an implementation using the pooling strategy.

Another idea that is currently missing in existing LSH implementations is the use of sketches, i.e., small representations of the original data points that allow to estimate the distance between two data points via their sketches. Christiani [12] describes how to use sketches when solving the near neighbor problem, but we are not aware of existing LSH-based implementations using this idea. We remark that sketching is a well-known technique and refer to the survey [27].

Auto Tuning Approaches. Apart from the approaches mentioned above, FLANN [26] and the implementation of vantage point trees in nmslib [8] are two non-LSH based nearest neighbor search that promise to tune the data structure to guarantee a certain quality criterion. This criterion is usually the recall of the query, i.e., the fraction of true nearest neighbor among the points returned by the implementation.

FLANN contains a collection of tree-based methods. For auto-tuning, it takes a small sample of the data structure and builds indexes in a certain parameter space. It then queries the data structure with points from the data set and picks, among all the indexes that achieve at least the recall the user wishes for, the one with fastest query times. The auto tuning employed by nmslib for the vantage point tree implementation follows the same principles and explores a certain parameter space based on a model of the data set to be indexed. Both approaches require that the query and data set distribution are not too different. We will see in the experiments that both of the approaches do not satisfy the recall guarantees, even on real-world datasets.

2 Preliminaries

2.1 Problem Definition

We assume a distance space $(\mathsf{X}, \mathrm{dist})$ with distance measure $\mathrm{dist}\colon \mathsf{X} \times \mathsf{X} \to \mathbb{R}_{\geq 0}$.

▶ **Definition 1.** *Given a dataset $S \subseteq \mathsf{X}$ and an integer $k \geq 1$, the (k, δ) nearest neighbor problem ((k, δ)-NN) is to build a data structure, such that for every query $q \in \mathsf{X}$, the query algorithm returns a set of k distinct points, each one being with probability at least $1 - \delta$ among the k points in S closest to q.*

An algorithm solving the (k, δ)-NN problem guarantees an expected recall of $(1 - \delta)k$, which is usually the quality measure in the context of nearest neighbor search algorithms.

2.2 Locality-Sensitive Hashing

▶ **Definition 2** (LSH Family [20, 10])**.** *A* locality-sensitive hash (LSH) family $\mathcal{H}$ *is family of functions $h\colon X \to R$, such that for each pair $x, y \in X$ and a random $h \in \mathcal{H}$, for arbitrary $q \in X$, whenever $dist(q, x) \leq dist(q, y)$ we have $p(q, x) := \Pr[h(q) = h(x)] \geq \Pr[h(q) = h(y)]$.*

Traditionally, LSH families are used in the LSH framework to solve the (c, r)-near neighbor problem.

While the theory provided in this paper applies to every distance space that encompasses an LSH family, we set our focus on solving the k-NN problem on the d-dimensional unit sphere under angular distance, which is equivalent to cosine similarity and inner product similarity on unit length vectors. By using the Gaussian kernel approximation method of Rahimi and Recht [29] as described by Christiani in [11], our results extend to the whole Euclidean space.

Random hyperplane (HP) LSH described by Charikar in [10] and Cross-Polytope (CP) LSH introduced by Terasawa and Tanaka [31] and analyzed by Andoni et al. in [1] are two different LSH schemes under this distance measure. A single HP LSH function produces a single bit. It works by choosing a random d-dimensional vector $a = (a_1, \ldots, a_d)$ where each a_i is an independent standard normal random variable. The hash code of a point x is 1 if the inner product between a and x is at least 0, and 0 otherwise. A single CP LSH function applies a random rotation of x on the unit sphere and then maps it to the index of the closest vector out of the $2d$ signed standard basis vectors. Technically, it can be thought of as choosing d random hyperplanes $a_1, \ldots, a_d$ and mapping x to the index of the hyperplane with the largest absolute inner product, separating the two cases that the inner product is negative or not.

3 Data Structure

This section describes the basic ideas of the data structure used in our implementation. Due to space reasons we only highlight the basic data structure and some of its properties. The full description with all proofs can be found in the extended version of this paper. Note that the implementation has many differences to this clean version. These differences are discussed in Section 4.

3.1 Description

In this section we will assume that we can perform distance computations and evaluate locality-sensitive hash functions in constant time. Our data structure will be parameterized by integers $L, K \geq 1$ and will consist of a collection of L LSH tries of max depth K. This data structure is known as an LSH Forest [5]. Here we use a variant of the LSH tries with bounded depth and for completeness we include a brief description and statement of relevant properties.

LSH tries. We index the LSH tries by $j = 1, \ldots, L$. The jth LSH trie is built from the set of strings $\{(h_{1,j}(x), \ldots, h_{K,j}(x)) \mid x \in S\}$ where $h_{i,j} \sim \mathcal{H}$. The trie is constructed by recursively splitting the set of points S on the next (ith) character until $|S| \leq i$ or $i = K + 1$ at which point we create a leaf node in the trie that stores references to the points in S. Internal nodes store pointers to its children in a hash table where the keys are locality-sensitive hash values.

Query algorithm. Let $S_{i,j}(q)$ denote the subset of points in S that collide with q when we consider the first i hash values used in the construction of the jth trie. That is, $S_{i,j}(q) = \{x \in S \mid h_{1,j}(q) = h_{1,j}(x) \wedge \cdots \wedge h_{i,j}(q) = h_{i,j}(x)\}$. For our query algorithm we wish to retrieve the points in each trie that collide with our query point in a bottom-up fashion, starting at depth $i = K$. Define $\Pi_{i,j}(q) = S_{i,j}(q) \setminus S_{i+1,j}(q)$ where $S_{K+1,j}(q) = \emptyset$.

▶ **Fact 1.** *LSH tries have expected construction time $O(nK)$ and use $O(n)$ words of space. For $i \in \{0, 1, \ldots, K\}$ we can retrieve a set $S'_{i,j}(q) \supseteq S_{i,j}(q)$ with $|S'_{i,j}(q)| \leq |S_{i,j}(q)| + i$ using time $O(|S_{i,j}(q)| + i)$. After having retrieved $S'_{i,j}(q)$ we can retrieve $S'_{i-1,j}(q)$ using additional time $O(|\Pi_{i-1,j}(q)|)$.*

The query algorithm is described in Algorithm 1. During a query we search through the LSH Forest starting with the buckets $S_{i,j}(q)$ at depth $i = K$ and moving up one level once the L LSH tries have been explored at the current level. While searching we use a data structure PQ to keep track of the top-k closest points seen so far. We stop the search once

Algorithm 1 ADAPTIVE-KNN(q, k, δ).

```
1  PQ ← empty priority queue of (point, dist) of unique points
2  for i ← K, K − 1, . . . , 0 do
3      for j ← 1, 2, . . . , L do
4          for x ∈ Π_{i,j}(q) do
               /* We abbreviate x'_k ← PQ.max() for ease of notation        */
5              if dist(q, x'_k) ≥ dist(q, x) then
6                  PQ.insert(x, dist(q, x))  // Remove largest entry if PQ contains
                    more than k elements.
7              end
8          end
9          if i = 0 or (PQ.size() == k and j ≥ ln(1/δ)/p(q, x'_k)^i) then
10             return PQ
11         end
12     end
13 end
```

we have searched sufficiently many tries at a depth where our current k-nearest neighbor candidate would have been found with probability at least $1 - \delta$. This stopping criterion ensures that we always search far enough to find the true k-nearest neighbor with probability at least $1 - \delta$.

Insertions (Line 6) and retrieval of the k-largest distance (Lines 5 and 9) can be done in expected amortized time $O(1)$ by using an array of size $2k$ which is updated after each k insertions. For simplicity the algorithm is described as a double for-loop that iterates over the sets $\Pi_{i,j}(q)$ in a bottom-up fashion. Using LSH tries Algorithm 1 would be implemented by using a straight-forward bottom-up traversal of the tries with properties described in Fact 1.

We proceed by proving that Algorithm 1 solves the (k, δ)-NN problem as well as providing running time bounds. The idea behind an adaptive kNN algorithm with guarantees following the approach of Algorithm 1 can be attributed to Dong et al. [18]. Christiani et al. [13] show how a different stopping criteria gives a self-tuning algorithm in the regime where $\delta = 1/n$. To the best of our knowledge both the following proof of correctness (although simple) and the running time bound for Algorithm 1 is new.

▶ **Lemma 3.** ADAPTIVE-KNN(q, k, δ) *returns a set of k points, each one being with probability at least $1 - \delta$ among the closest k points to q.*

Proof. As introduced in Definition 2, we use the short notation $p(q, x) := \Pr[h(q) = h(x)]$ under the random LSH hash function choice h. Let $x_1, \ldots, x_k$ be the k closest neighbors of q in S. First, observe that at any stage the algorithm maintains the invariant "PQ.size() $<$ k **or** $p(q, \mathrm{x}'_\mathrm{k}) \leq p(q, x_k)$". This is true because at any point, the k-th closest point x'_k identified by the algorithm satisfies $\mathrm{dist}(q, x'_k) \geq \mathrm{dist}(q, x_k)$. Together with the montonicity of the collision probability of LSH, cf. Definition 2, the invariant holds. Thus, the algorithm cannot terminate until j' tries have been searched at level i' where either $j' \geq \ln(1/\delta)/p(q, x_k)^{i'}$ or $i' = 0$. In the first case the probability of not finding a kNN of q is at most $(1-p(q, x_k)^{i'})^{j'} \leq \delta$. In the case of $i' = 0$ the query algorithm degrades to a linear scan and we are guaranteed to report the true kNNs. ◀

Next, we connect the expected running time of Algorithm 1 to the optimal expected running time of an algorithm that knows optimal parameter choices for i and j. We will use $OPT(L, K, k, \delta)$ to denote the optimal expected query time that can be achieved with the natural algorithm that solves kNN queries on the LSH Forest by searching j tries at depth i where i and j are chosen to minimize the query time. In our expression for the expected the query time we use a unit cost model that counts hash function evaluations and distance computations. To ensure that each point in the kNN set is reported with probability at least $1 - \delta$ we search $j = \ln(1/\delta)/p(q, x_k)^i$ tries. The expected cost of searching one LSH trie at depth i is $i + \sum_{x \in P} p(q, x)^i$.

$$OPT(L, K, k, \delta) = \min \left\{ \frac{\ln(1/\delta)}{p(q, x_k)^i}(i + \sum_{x \in P} p(q, x)^i) \mid 0 \leq i \leq K, \frac{\ln(1/\delta)}{p(q, x_k)^i} \geq L \right\}.$$

We obtain the following lemma with a proof provided in the extended version of this paper.

▶ **Lemma 4.** *Let $0 < \delta \leq 1/2$ then with probability $1-\delta$ we have that* ADAPTIVE-KNN(q, k, δ) *terminates in expected time $O(OPT(L, K, k, \delta/k) + L(K + k))$.*

3.2 Reducing the Number of LSH Evaluations

In [12], Christiani provides a uniform framework encompassing previous ad-hoc solutions [15, 3] to reduce the amount of hash function evaluations when solving the approximate near neighbor problem. In the extended version of this paper we describe how these techniques can be applied when solving the k-NN problem. In particular, each method requires an adaption of Algorithm 1 and comes with their own stopping criterion. We provide a succinct description of the methods next.

Tensoring. Assume that K is an even integer and L is an even power of two. Form two collections of $\sqrt{L}$ tuples of $K/2$ LSH functions. Each trie in the LSH forest is now indexed by $j_1, j_2 \in \{1, \ldots, \sqrt{L}\}$. The K LSH functions used in the (j_1, j_2)st trie are taken by interleaving the $K/2$ functions in the j_1st tuple of the first collection and the j_2nd tuple of the second collection. This allows us to construct L LSH tries of max depth K using only $\sqrt{L}K$ independent functions.

Pooling. Form a pool of m independent LSH functions that will be shared among LSH tries. For each LSH trie in the LSH Forest we independently sample a random subset of K LSH functions from the pool that we use in place of fully random LSH functions. The LSH pool can be viewed as a randomized construction of a smaller LSH family from our original LSH family. As m increases LSH functions sampled without replacement from the pool will work almost as well as independent samples from the LSH family.

3.3 Sketching for faster distance computations

Locality-sensitive hashing can be used to produce 1-bit sketches for efficient similarity estimation [10, 23]. The idea is to use a random hash function to hash the output of a locality-sensitive hash function to a single bit, and then packing w such bits into a w-bit machine word. We can use word parallelism (alternatively table lookups) to count the number of collisions between $w = \Theta(\log n)$ such sketches in $O(1)$ time, allowing us to efficiently estimate the similarity between points in our original space. Depending on the LSH scheme used and the distribution of distances between the query point and the data, using 1-bit sketches can replace many of the expensive distance computations performed by the query algorithm with cheaper distance estimations through sketching. See [30, 12] for more details on sketching in the context of the ANN problem.

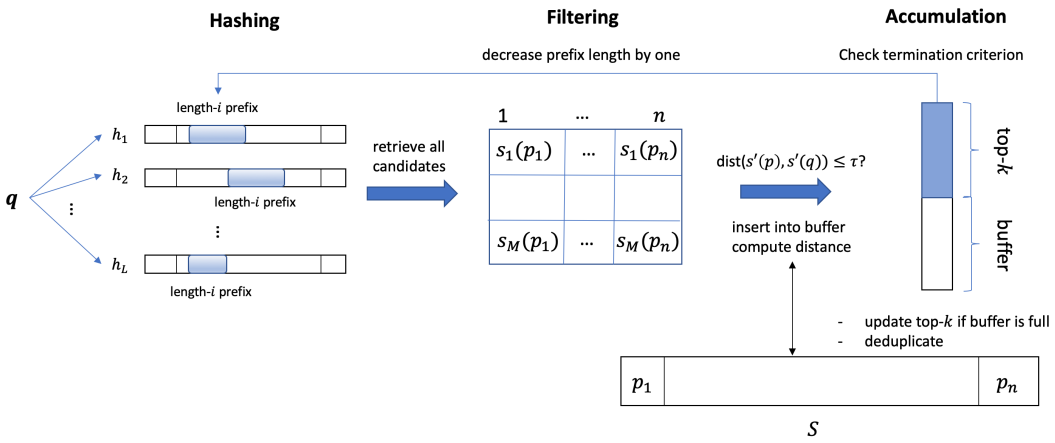

Figure 1 Overall structure of our implementation.

4 Implementation Overview

4.1 Overall structure

Figure 1 presents an overview over our data structure. Deviating from the pointer-based trie data structure described in Section 3, we use an array of indices sorted by hash code, which improves both cache- and space-efficiency. Additionally, a sketching-based filtering approach is used to reduce the number of distance computations carried out during a query. In the following, we make the implementation precise.

A query is answered in rounds, starting at the maximum considered prefix length. A single round works as follows: Repetitions are inspected one after the other. In each repetition, all (new) candidates sharing the hash prefix with the query point are retrieved. For each such candidate point, a sketch is chosen and checked against the corresponding sketch of the query point. Let τ denote a threshold value that we will discuss how to set later. If the Hamming distance between candidate and query sketch is less than τ, the data point passes the filter, the distance computation is carried out, and the point with its distance to the query is inserted into the accumulator buffer. Once this buffer is full we discard all points not belonging to the top-k. While this can be done in time $O(k)$, we found that an implementation based on sorting was faster for the values of k we considered. The termination criterion is checked after all repetitions are inspected. If the criterion is satisfied, the algorithm returns the indices of the top-k points in the accumulator, otherwise the prefix length is decreased by one and a new round starts.

The following subsections make this process more detailed and discuss engineering choices in the case of angular distance on unit length vectors.

4.2 Engineering choices

Vector storage. We normalize the query vector and all data vectors, which means that all dot products will be between -1 and 1. This allows storing vectors in a fixed point format represented using 16-bit integers. Such a representation enables AVX2-enabled instructions that allow 16 multiplications at once. To use the AVX instructions, all vectors are stored in a 1-dimensional array with padding to ensure 256-bit alignment.

Retrieving candidate points. The jth LSH repetition is represented as a sorted array of tuples of the form $(h_{\leq i,j}(p), \text{pos}(p))$, where $\text{pos}(p)$ is the index of $p \in S$ in the original dataset, and $h_{\leq i,j}(p)$ is the hash code of p under hash functions $h_{1,j} \circ \cdots \circ h_{i,j}$. We view the hash code as a bitstring. Before retrieving candidates, we first find the tuple with the longest common prefix with the hash of the query vector. This is achieved using binary search, where we tabulate the lexicographic position of each 13-bit prefix to speed up the search.

Each time more candidates are requested, all tuples whose hash code has a common prefix of length at least i are considered. Each iteration decrements i in order to increase the number of considered vectors. Since the vectors are stored in sorted order, all tuples with a common prefix of length i are stored adjacent to the tuples with common prefixes of length $i + 1$. Furthermore, they are all stored on the same side, depending on whether the removed bit was a 0 or 1. This means that the range of considered vectors can be updated efficiently.

Every access in the array is done in a segment of size $B = 12$, regardless of whether the prefix matches or not. This costs almost no time, because the random memory access is the expensive part, and only improves quality. A discussion of suitable values of B is provided in the extended version of this paper.

Filtering candidate points. The filtering step is an additional measure to reduce the number of distance computations. Fix a point $p \in S$. During the index building phase, we store M 64-bit sketches $s_1(p), \ldots, s_M(p)$ obtained via HP LSH. If p is retrieved as a candidate, retrieve a sketch $s'(p)$ using a pseudorandom transformation of the repetition number j. Next, compute the Hamming distance between $s'(p)$ and $s'(q)$. If the distance is at most $\tau \in \{0, \ldots, b\}$, p passes the filter and is inserted into the accumulator. A challenge in the context of k-NN queries is that the algorithm does not know the distance to the k-th nearest neighbor. This means that the threshold has to be adapted according to the points inspected so far. We set threshold τ dynamically according to the probability that a vector with Hamming distance τ or less has a dot product larger than the smallest dot product in the current top-k.

Computing distances. The accumulator takes care of the candidate points that pass the filter step. It de-duplicates the candidate list and keeps track of the top-k points found so far. The accumulator consists of a buffer of size $2k$, which contains the current top-k indices, along with their dot products, and a buffer of size k, which contains points that passed the filter along with their dot products. Once this buffer is full, the top-k list is updated.

4.3 Locality-sensitive Hash Functions

Supported LSH Functions. The supported hash functions relevant for the paper are HP LSH [10] and CP LSH [1].[1] For the latter, the implementation encompasses both the exact version and the pseudorandom version with three applications of the fast Hadamard transform, see [1] for more details. We always regard hash functions as producing an ℓ-bit string as its output. For HP LSH, we have $\ell = 1$, for CP LSH, we have $\ell = \lceil \log 2d \rceil$. In case the algorithm did not terminate after exploring all L repetitions, decreasing the prefix length by one always means that we disregard the last bit of the hash. This is to avoid a sudden increase in the number of collisions in the case of CP LSH. This is theoretically sound since the termination criteria from Section 3 only need a lower bound on the collision probability at a certain prefix-length, which can be estimated for individually bit lengths of CP LSH.

[1] PUFFINN is generic to the LSH family, but some engineering choices are different for other similarity measures, such as set similarity. At the moment, PUFFINN also support (b-bit) MinHash [9, 23].

Estimating Collision Probabilities. Recall from Section 3 that evaluating the collision probability of two points at a certain distance is a key ingredient in the query algorithm. While such a formula is easy to derive for HP LSH, we only know of the asymptotic behavior of collision probabilities for CP LSH [1]. To overcome this obstacle, we find a Monte Carlo estimate on the collision probability of unit vectors with inner product $\alpha, -1 \leq \alpha \leq 1$, by enumerating different values of α in a window of size .05. For a fixed distance, we consider two points $x = (1, 0, \ldots, 0)$ and $y = (\alpha, \sqrt{1-\alpha^2}, 0, \ldots, 0)$,[2] draw 1 000 random CP hash functions, count the number of collisions, and tabulate the estimate. As mentioned above, we always consider bit strings, so the probability estimation for CP LSH is made for all bit lengths up to $\ell = \lceil \log 2d \rceil$.

In the query procedure, we round the distance down to the closest distance value for which we have tabulated an estimate and use that to bound the collision probability. The evaluation in the next section will show that this yields a negligible loss in quality compared to an exact variant using HP LSH.

5 Experimental Evaluation

Implementation and Experimental Setup. PUFFINN is implemented in C++ and comes with a wrapper to the Python language. Experiments were run on 2x 14-core Intel Xeon E5-2690v4 (2.60GHz) with 512GB of RAM using Ubuntu 16.10 with kernel 4.4.0. It is compiled using `g++` with the compiler flags `-std=c++14 -Wall -Wextra -Wno-noexcept-type -march=native -O3 -g -fopenmp`. Index building was multi-threaded, queries were answered sequentially in a single thread. The experiments were conducted in the `ann-benchmarks` framework from [4]. The code, raw experimental results, and the Jupyter notebook used for the evaluation are available at `https://github.com/puffinn/esa-paper`.

Quality and Performance Metrics. As quality metric we measure the individual recall of each query, i.e., the fraction of points reported by the implementation that are among the true k-NN. As performance metric, we record individual query times. We usually report on the *throughput*, i.e., the average number of queries that can be answered in one second. In plots, the throughput is dubbed QPS for *queries per second*.

Parameter Choices. PUFFINN has two parameters: the space a user is willing to allocate for the index, and the expected recall that should be achieved. We run PUFFINN with expected recall values in the set $\{.1, .2, .5, .7, .9, .95\}$. As space parameters, we use the doubling range 512 MB to 32 GB. We always retrieve the ten nearest neighbors.

Objectives of the Experiments. Our experiments are tailored to answer the following high-level questions (*HL-Q*):

(HL-Q1) Given the choices of sketching and hash function evaluation methods described in the previous two sections, how do they compare to each other w.r.t. empirical performance (Sections 5.1–5.3)?

(HL-Q2) Can a parameterless method compete with implementations that have parameters tuned to the data and query workload (Section 5.5)?

[2] By spherical symmetry, the collision probabilities are the same for all pairs of points with inner product α.

Table 1 Datasets under consideration.

Dataset	Data Points/Query Points	Dimensions
GLOVE [28]	1 183 514/10 000	100
GLOVE-2M [28]	2 196 018/10 000	300
GNEWS-3M [25]	3 000 000/10 000	300
SYNTHETIC	1 000 000/ 1 000	300

To answer these questions, we will consider the following implementation-level questions (*IL-Q*):

(IL-Q1) What is the influence of the filtering approach to the quality/QPS (Section 5.1)?

(IL-Q2) What is the influence of the update threshold τ to quality and QPS (Section 5.1)?

(IL-Q3) How does the space parameter influence the QPS (Section 5.2)?

(IL-Q4) How does the hash function and evaluation strategy influence performance (Section 5.3)?

Real-World Datasets. Table 1 gives an overview of the datasets used in the experiments. Cosine similarity is usually used in the context of word embeddings, so we use three real-world datasets that originate from two different word embedding algorithms. Unless stated otherwise, all experiments are carried out on Glove-1M.

Synthetic Dataset. We describe a synthetic data set and query distribution that, as we will see, is challenging for many heuristic nearest neighbor implementations. For a fixed $d \geq 1$, we construct a dataset over $\mathbb{R}^{3d}$ as follows. For each $i \in \{1, \ldots, n-1\}$, let y_i and z_i be two d-dimensional vectors of expected length $\sqrt{1/2}$ where each coordinate is sampled independently from $\mathcal{N}(0, 1/2d)$. Define $x_i = (0^d, y_i, z_i)$. Let v and w be two more random vectors of expected length $\sqrt{1/2}$. Finally, set $x_n = (v, w, 0^d)$. We define m query vectors as follows: For each $i \in \{1, \ldots, m\}$, let $q_i = (v, 0^d, r_i)$, where r_i is a random vector of length $\sqrt{1/2}$.

This construction has the property that x_n is the nearest neighbor of every query. Furthermore, all data points have unit length in expectation. The distance from each q_i to x_n is expected to be 1 (or equivalently $\mathrm{E}[\langle q_i, x_n \rangle] = \frac{1}{2}$), whereas the distance to all other points is around $\sqrt{2}$ (i.e., $\mathrm{E}[\langle q_i, x_j \rangle] = 0$). In the experiment, we choose $n = 1\,000\,000$ and $d = 100$.

Other approaches. We compare PUFFINN to the following implementations: FALCONN, a state-of-the-art LSH implementation using the theory developed in [1]; ONNG, a recent graph-based approach described in [21]; ANNOY, the best-performing implementation of a random-projection forest [7]; IVF, a k-means clustering based approach [22]; FLANN, a collection of different approaches with tuning of recall value [26]; VantagePointTree [32] as implemented in NMSlib [8] with recall guarantees.

These approaches stood out in the benchmarking paper from Aumüller et al. [4] as performing best on many datasets. We use the same parameter space as in [4] to test the performance of the different implementations. For each implementation, we report the best results achieved via a grid search over the (usually large) parameter space. We refer to that paper or the original papers for more details on these approaches. Except from FLANN and VantagePointTree, no other implementation allows to specify a guarantee on recall.

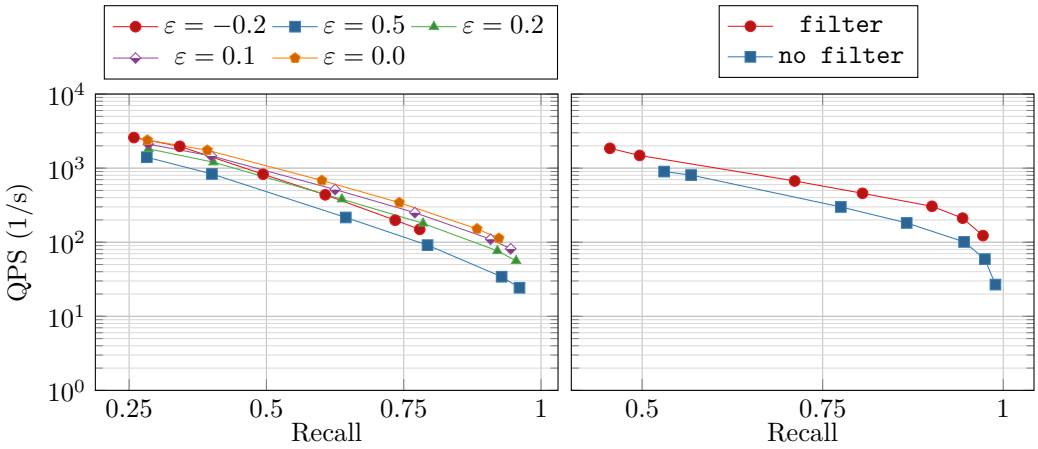

Figure 2 Left: Influence of setting the threshold of sketches to $1+\varepsilon$ times the expected difference at the distance of the k-th closest point found so far. Expected recall values: 0.1, 0.2, 0.5, 0.7, 0.9, 0.95, space: 1GB. Right: Difference between filtering/no filtering, space: 4GB.

5.1 Filtering Approach

We evaluate the filtering approach in two directions. First, we report on the quality-performance trade-off of different update strategies. Second, we benchmark the architecture against a "no filtering" approach. Experiments were done using a collection of 32 sketches using 64 bits each.

Figure 2 (top) reports on the influence of setting the passing threshold of the filtering step dynamically to a fraction of $\tau = 1 + \varepsilon$ of the expected difference[3] for a point at the distance of the current k-th nearest neighbor. A ε-value of $0.0, 0.1$, and 0.2 give good results in this empirical setting. Above 0.1 there is almost no gain in quality but a huge drop in QPS. Setting the threshold below the expectation results in a large loss in quality. For the remainder of the experiments, we set the threshold to the expectation, i.e., we use $\delta = 0$.

Figure 2 (bottom) allows us to see the difference between using resp. not using the filtering approach. For low recall values, the filtering approach increases the QPS by a factor of roughly 1.5. For example, the filtering approach can answer around 1 400 QPS at recall .5, without filtering this number drops to 900. At high recall, the difference is more pronounced. A recall of 97% is achieved with 122 QPS using filtering and the same recall is achieved at around 50 QPS without filtering. We can see a clear difference in the achieved quality between the two variants. The sketching approach usually decreases the recall for a fixed expected recall by .08 (for low recall) to .03 (for high recall). However, the recall is still above the guarantee in both cases.

5.2 Influence of the Index Size

We turn our focus to the third implementation-level question: How does the index size influence the performance? We note that the index size includes the whole data structure, including storage of the original dataset and the hash functions. Figure 3 reports on the quality-performance trade-off achieved by the implementation for different index sizes. We

[3] The expected difference is just 64 times the probability that two points at distance r_k collide under a random hyperplane.

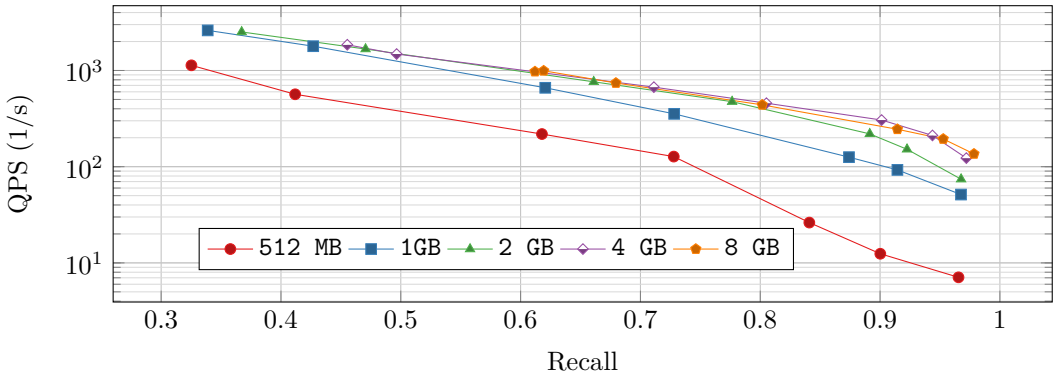

Figure 3 Influence of index size to quality-performance trade-off.

observe that larger index sizes provide better performance, but the influence is diminishing at more than 4 GB. A small index yields a small number of repetitions, which means that the algorithm has to explore many levels in the data structure. For a recall of .9, increasing the amount of space from 512 MB to 2 GB increases the QPS from 12 to roughly 200. Doubling the allotted space to 4GB results in around 300 QPS, which is roughly the same for 8 GB as well. We can see that the achieved recall is above the set guarantee threshold for all tested index sizes. (Each data point corresponds to a recall value from the set of tested recall values.)

5.3 Choice of Hash Function and Evaluation Strategy

We start by evaluating different hash evaluation strategies. We implemented the following three different evaluation strategies in PUFFINN: independent, tensor, and pool. For the pooling strategy, we set up a pool containing 3072 bits. In the following, we fix the hash function used to be CP LSH using fast Hadamard transform. Figure 4 shows a comparison between the three evaluation strategies for different index sizes. As we can see, tensoring is never better than the pooling strategy. Furthermore, independent gives better performance for a fixed quality for large index sizes, but takes more time for initializing the hash values at low recall. We note that when using the exact CP LSH, there is a huge difference between independent and pooling, in particular for large index sizes. For example, CP using independent hash functions achieves not more than 80 QPS for the 8 GB index in the right plot in Figure 4. For all of these reasons, we fix the implementation to use the pooling strategy.

We turn our focus to the choice of hash function. Figure 5 gives an overview of three different index sizes using the pooling strategy for hash function evaluation. On the smallest index, HP LSH works well, in particular for high recall. If there is space for more repetitions, CP LSH becomes the method of choice, and FHT-CP is a bit faster than the exact method.

5.4 Summary of Implementation Choices

In light of our first high-level question, we observed that sketching provides a good performance increase at negligible cost. The larger the index size, the faster PUFFINN can answer queries. However, it works well on small index sizes as well. We fixed the pooling strategy as the evaluation strategy since it was much faster than tensoring and allows to use exact methods such as CP LSH when FHT-CP does not satisfy the guarantees one wishes for.

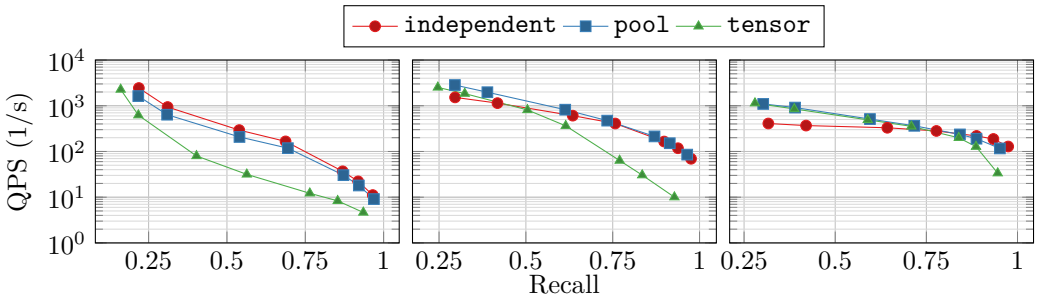

Figure 4 Comparison of hash evaluation strategies. Left: 512 MB, center: 2 GB, right: 8 GB.

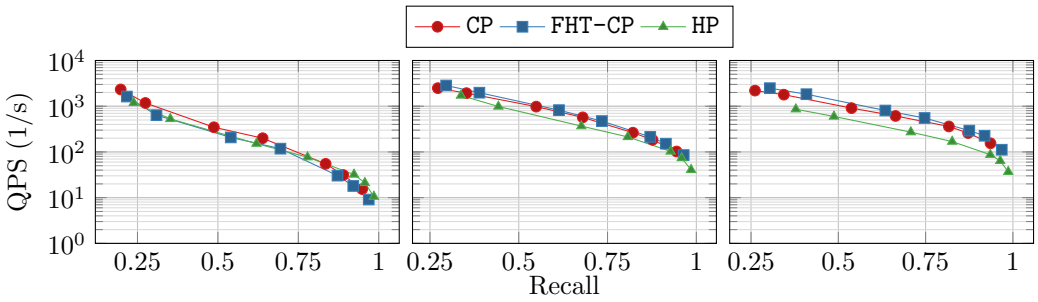

Figure 5 Hash function comparison (pooling). Left: 512 MB, center: 1 GB, right: 4 GB.

5.5 Comparison to Other Approaches

We turn our focus on comparing PUFFINN to other approaches on real-world and synthetic datasets (HL-Q2). Figure 6 gives an overview of the performance-quality trade-off achieved by state-of-the-art k-NN approaches on three real-world dataset and a synthetic one. Among the implementations that support automatic parameter tuning, PUFFINN is usually by a large factor the fastest implementation. We remark that the automatic parameter tuning of FLANN failed to build an index within 6 hours on three out of the four datasets and was disregarded on those.

PUFFINN managed to obtain at least recall .95 on every dataset, whereas the tuning of the VPTree failed to achieve high recall on GNEWS-3M. PUFFINN shows better performance than FALCONN for most of the performance-quality space. It is comparable in performance to ANNOY on most of the real-world datasets, and is particular competitive in the high-recall setting. On real-world datasets, IVF and in particular ONNG show better performance than PUFFINN except for very high recall. This is an indicator that graph-based approaches perform best on these real-world datasets (for good manual parameter choices).

On the synthetic dataset, only LSH-based approaches achieve high recall at high QPS. VPTree and ONNG manage recall close to 1, but are more than a factor 10 slower than PUFFINN. IVF and ANNOY fail to achieve recall higher than 40% on the synthetic dataset.

To come back to our second high-level question: It is possible to compete with state-of-the-art implementations of k-NN using a parameterless method with guarantees. PUFFINN is easy to use and achieves performance comparable to many of its competitors. Among LSH variants, it is as fast as FALCONN which does not give guarantees. This means that our engineering choices allowed an LSH implementation with theoretical guarantees that come “for free”.

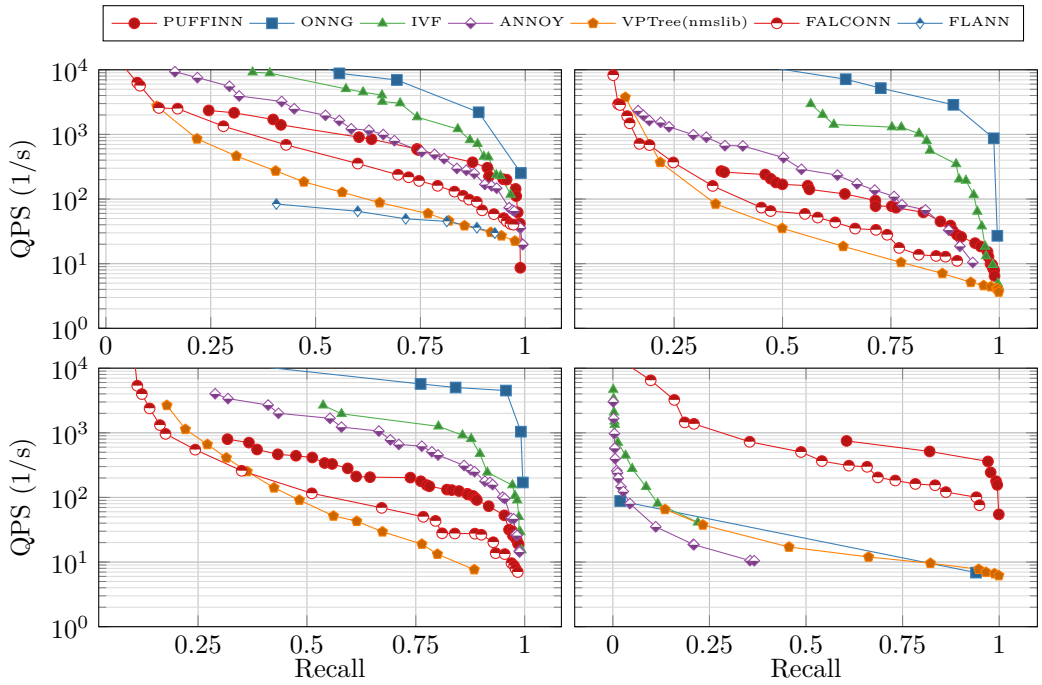

Figure 6 Comparison of different implementations with PUFFINN (index size at most 8 GB). Top left: Glove-1M, top right: Glove-2M, bottom left: GNEWS-3M, bottom right: Synthetic.

References

1. A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. In *Proc. NIPS '15*, pages 1225–1233, 2015.
2. Alexandr Andoni and Piotr Indyk. E2LSH, user manual, 2005.
3. Alexandr Andoni and Piotr Indyk. Efficient algorithms for substring near neighbor problem. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1203–1212. Society for Industrial and Applied Mathematics, 2006.
4. Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. In *SISAP'17*, pages 34–49, 2017. doi:10.1007/978-3-319-68474-1_3.
5. Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of 14th International Conference on World Wide Web (WWW)*, pages 651–660. ACM, 2005.
6. Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.
7. Erik Bernhardsson. Annoy. URL: https://github.com/spotify/annoy.
8. Leonid Boytsov and Bilegsaikhan Naidan. Engineering Efficient and Effective Non-metric Space Library. In *SISAP'13*, pages 280–293, 2013. doi:10.1007/978-3-642-41062-8_28.
9. Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.
10. Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of 34th ACM Symposium on Theory of Computing (STOC)*, pages 380–388, 2002. doi:10.1145/509907.509965.
11. Tobias Christiani. A Framework for Similarity Search with Space-Time Tradeoffs using Locality-Sensitive Filtering. In *Proc. 28th Symposium on Discrete Algorithms (SODA)*, pages 31–46, 2017.

12 Tobias Christiani. Fast Locality-Sensitive Hashing for Approximate Near Neighbor Search. *CoRR*, abs/1708.07586, 2017. arXiv:1708.07586.

13 Tobias Christiani, Rasmus Pagh, and Mikkel Thorup. Confirmation Sampling for Exact Nearest Neighbor Search. *CoRR*, abs/1812.02603, 2018. arXiv:1812.02603.

14 Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97*, pages 426–435, 1997. URL: http://www.vldb.org/conf/1997/P426.PDF.

15 Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. Fast Similarity Sketching. *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 663–671, 2017.

16 Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *STOC'08*, pages 537–546. ACM, 2008.

17 Wei Dong. LSHkit. URL: http://lshkit.sourceforge.net/.

18 Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling LSH for performance tuning. In *Proceedings of 17th ACM Conference on Information and Knowledge Management (CIKM)*, pages 669–678. ACM, 2008.

19 Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality. *Theory of Computing*, 8(1):321–350, 2012. doi:10.4086/toc.2012.v008a014.

20 Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of 30th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 604–613, 1998. doi:10.1145/276698.276876.

21 M. Iwasaki and D. Miyazaki. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *ArXiv e-prints*, October 2018. arXiv:1810.07355.

22 Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *CoRR*, abs/1702.08734, 2017. arXiv:1702.08734.

23 Ping Li and Arnd Christian König. Theory and applications of b-bit minwise hashing. *Commun. ACM*, 54(8):101–109, 2011. doi:10.1145/1978542.1978566.

24 Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *ArXiv e-prints*, March 2016. arXiv:1603.09320.

25 Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR*, abs/1301.3781, 2013. arXiv:1301.3781.

26 Marius Muja and David G. Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *VISSAPP'09*, pages 331–340. INSTICC Press, 2009.

27 Rasmus Pagh. Similarity Sketching. In *Encyclopedia of Big Data Technologies.* Springer, 2019. doi:10.1007/978-3-319-63962-8_58-1.

28 Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

29 Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2008.

30 Venu Satuluri and Srinivasan Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *Proceedings of the VLDB Endowment*, 5(5):430–441, 2012.

31 Kengo Terasawa and Yuzuru Tanaka. Spherical LSH for Approximate Nearest Neighbor Search on Unit Hypersphere. In *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*, pages 27–38, 2007. doi:10.1007/978-3-540-73951-7_4.

32 Peter N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA.*, pages 311–321, 1993. URL: http://dl.acm.org/citation.cfm?id=313559.313789.

33 Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. Approximate Similarity Retrieval with M-Trees. *VLDB J.*, 7(4):275–293, 1998. doi:10.1007/s007780050069.