Solving k-Closest Pairs in High-Dimensional Data

Martin Aumüller$^{1[0000-0002-7212-6476]}$ and Matteo Ceccarello$^{2[0000-0003-2783-0218]}$

^{1} IT University of Copenhagen, Denmark
maau@itu.dk
^{2} University of Padova, Italy
matteo.ceccarello@unipd.it

Abstract. We investigate the k-closest pair problem in high dimensions, that is finding the $k \geq 1$ closest pairs of points in a set $S \subseteq \mathcal{X}$ in a metric space $(\mathcal{X}, \mathrm{dist})$. This is a fundamental problem in computational geometry with a wide variety of applications, including network science, data mining, databases, and recommender systems. We propose an *exact* algorithm with a controllable *failure probability*, thus allowing the user to specify the desired *recall*. Our algorithm has expected *subquadratic* running time under mild assumption on the distance distribution, relying only on the existence of a Locality Sensitive Hash family for the metric at hand. We complement our theoretical analysis with an experimental evaluation, showing that our approach can provide solutions orders of magnitude faster than current state-of-the-art data structures designed for specific metrics.

1 Introduction

In this paper we study the k-*closest pair problem*: Given a set $S \subseteq \mathcal{X}$ from a metric space $(\mathcal{X}, \mathrm{dist})$, the task is to identify k pairs of distinct points in $S \times S$ that are closest to each other. Solving this problem has numerous applications, for example in network science [29], data mining [22], and databases [21].

A naïve solution is to employ an all-to-all comparison between the points in S. This will result in $O(|S|^2)$ distance comparisons; if S has n elements, the running time will thus become *quadratic*. For metrics such as *Manhattan* distance or *Euclidean* distance, there exist approaches for solving the closest pair problem in d dimensions in time $2^{O(d)} n \log^{O(d)} n$ [13,8], which lead to subquadratic running times if the dimensionality d is small. However, these data structures suffer from the well-known curse of dimensionality because they have an *exponential dependence* on d.

To design *scalable* closest pair algorithms with *subquadratic running time guarantees*, research settled on allowing the results to be *approximate*. In a strong theoretical sense, this means that if the closest pair is at distance r, then an algorithm guarantees to return a pair at distance at most cr for some approximation factor $c > 1$ (with some small failure probability). As we will discuss in the related work section, many industry-strength solutions use the word more loosely to refer to the inexactness of results. The generally accepted measure of the quality of such approaches is their *recall*, i.e., the fraction of correct pairs identified by the algorithm. These solutions usually do not give strong guarantees on this quality measure. In this work, we propose a Locality-Sensitive Hashing [18] (LSH) based solution with theoretical guarantees on the expected achieved recall. In practice, this means that users only set two parameters: The amount of memory available for the index, and the recall guarantee.

In this work, we propose an extension of the LSH-forest approach by Bawa et al. [5] in the variant described by Aumüller et al. in [4]. In the latter work, the authors describe a query algorithm that carries out a bottom-up traversal of the LSH tries employed by the LSH forest with an adaptive stopping criterion. In Section 3, we will describe a traversal approach to solve the k-closest pairs problem. Intuitively, we first build build an LSH forest, which consists of L tries indexing the dataset vectors according to their LSH hash codes. The closest pairs are found by merging nodes in a bottom-up traversal of the trie, keeping track of the best candidate pairs found so far. We prove that our algorithms adapt well to the data distribution: Theorem 2 shows that up to some small additional cost factors, the running time of the proposed algorithm is *asymptotically equivalent* to an LSH-based *clairvoyant* algorithm that queries the part of the LSH forest that minimizes the expected work by knowing *the exact distance distribution.*

In Section 4 we describe implementation choices surrounding the proposed approach. For example, we store the trie as a flat array sorted by hash code to support efficient merging of subtrees in the trie, make use of sketches to save the expensive evaluation of distance computations, and discuss details of the parallelization strategy both for index building and the bottom-up traversal. The experimental evaluation is presented in Section 5 and compares our approach to several industry-standard baselines such as Meta Research' popular *FAISS* [19] library. We show the competitiveness of our approach to these approaches, in particular under the light that we compare the results to an intensive grid search for best parameters for the other approaches, whereas our approach just takes the available space and a recall guarantee as parameters.

Related Work

LSH approaches. Locality-sensitive hashing [18] is the de-facto standard for providing theoretically sound algorithms for the approximate near neighbor problem. Popular LSH functions include E2LSH [12] for Euclidean space, SimHash [9] and Crosspolytope LSH [2] for inner product similarity (or cosine similarity) on the unit sphere, and MinHash [6] and 1-bit MinHash [20] for set similarity under Jaccard similarity. Traditionally, the LSH framework aims to solve the (c, r)-near neighbor problem that requires to return a point at distance at most cr to a query point if there exists a point at distance at most r (with some constant probability). The k-NN problem can either be solved using a reduction to different (c, r)-near neighbor problem instances [16], or via direct approaches such as the LSH forest [5] and its variant [4] that we base our work on. In the database community, other directions to LSH-based indexing became very popular. These approaches use locality-sensitive hash functions to project the data points to a lower-dimensional space and index them using I/O-efficient tree data structures. For example, `LSBTree` [24] projects the points to a lower dimension using LSH, employs the Z-order to map points to a single value and indexes these values using B-Trees. A more recent approach called `PM-LSH` [28] indexes the projected points directly using the PM-Tree [23] without applying the Z-order.

Closest pair algorithms. One of the seminal papers on efficient solutions to the closest pair problem in high-dimensional data is by Xiao et al. [26]. It mainly focuses on Jaccard similarity but also discusses Cosine, Dice, and Overlap similarity. At a high level, it maintains the input sets in a priority queue ordered by an upper bound on the similarity it can attain with any other set, based on prefix filtering. Sets are extracted in decreasing order of such an upper bound and their similarity with other sets is computed by means of an inverted index on the tokens. Several optimizations to this approach were introduced in the recent paper [25]. Further improvements for sets under the *overlap* similarity are discussed in [27]. In particular, they propose a variant of [26] that evaluates more than one token for each point that is popped from the priority queue. While this approach improves the performance in the case of the overlap similarity, the authors discuss that for the Jaccard similarity it provides little benefit over [26].

For high-dimensional data using Euclidean distance, closest pairs can be found both by the `LSBTree` [24] and by the more recent `PM-LSH` [28]. In particular, `LSBTree` maintains a guess on the smallest k-th distance, and generates candidate pairs from the points whose difference of Z-values is below a threshold derived from the current guess. Another approach for the Euclidean distance was presented in [7]. Using random projections, points are mapped on the real line, where candidate pairs are generated from intervals of consecutive projections.

In general metric spaces, [15] provides a solution used on the *count M-tree* index, a variant of the classical M-tree data structure [11].

2 Preliminaries

Consider a metric space $(\mathcal{X}, \mathrm{dist})$, and let $k > 0$ be an input parameter. Let $S^2 := \{(s, s') \in S \times S \mid s \neq s'\}$ be the set of distinct pairs in S.

Definition 1. *The* k-closest pairs *in a set* $S \subseteq \mathcal{X}$ *are a sequence of distinct pairs* $(r_1, s_1), \ldots, (r_k, s_k) \in S^2$ *such that: For all other pairs* $(r, s) \in S^2$ *and for all* $i \in \{1, \ldots, k\}$, $\mathrm{dist}(r_i, s_i) \leq \mathrm{dist}(r, s)$.

Informally, the task is to find a set of k closest pairs of points in S. For $k = 1$, this problem is called the *closest pair problem.*

Naïvely, the problem can be solved by enumerating all pairs of points in S^2, leading to $O(|S|^2)$ distance comparisons.

In this paper we present randomized algorithms with probabilistic guarantees. This means that our algorithm receives two input parameters k and $\delta \in (0, 1)$.

If a pair $(r, s) \in S^2$ is a k-closest pair, then it is output by the algorithm with probability at least $1 - \delta$. If the quality of the solution is measured using *recall*, the fraction of correct pairs reported by the algorithm, we *expect* a recall of $1 - \delta$.

Definition 2 (Locality-Sensitive Hashing [18]). *Let $(\mathcal{X}, \text{dist})$ be a metric space, let T be a set, and let $\mathcal{H}$ be a family of functions $h \colon \mathcal{X} \to T$. For positive reals r_1, r_2, q_1, q_2, with $q_1 > q_2$, $\mathcal{H}$ is (r_1, r_2, q_1, q_2)-sensitive if for $x, y \in \mathcal{X}$ and h sampled uniformly at random from $\mathcal{H}$ we have that:*

- $\text{dist}(x, y) \leq r_1 \Rightarrow \Pr[h(x) = h(y)] \geq q_1$
- $\text{dist}(x, y) \geq r_2 \Rightarrow \Pr[h(x) = h(y)] \leq q_2$

As a technical detail, we assume that the LSH family is *monotonic*, i.e., its collision probability function is decreasing with the distance. Moreover, we assume that we can evaluate the probability of collision at a certain distance.[3] We denote the collision probability function with $p \colon \mathbb{R} \to [0, 1]$ and for ease of notation use $p(x, y) := p(\text{dist}(x, y))$ for $x, y \in \mathcal{X}$. Most popular LSH families have this property, such as Euclidean LSH [12] for Euclidean space, random hyperplane hashing [9] and Cross-Polytope hashing [2] for the d-dimensional unit sphere under inner product similarity (or cosine similarity), or 1-bit MinHash described by Li and König in [20] for set similarity under the Jaccard similarity. Since we use LSH functions as a black-box, our results hold for all LSH families that have this property and are not restricted to special cases.

In [4], Aumüller et al. introduced `PUFFINN`, a highly-optimized implementation of an LSH-based k-nearest neighbor search algorithm. Their work builds upon the LSH forest data structure of Bawa et al. [5] and the adaptive search mechanism described by Dong et al. in [14]. Since our work extends their data structure, we provide a recap of how their data structure works next. See [4] for more details.

Given a set $S \subseteq \mathcal{X}$, two parameters $L, K \geq 1$, and access to an LSH family $\mathcal{H}$, the data structure consists of a collection of L LSH tries of max depth K. The LSH tries are indexed by $j = 1, \ldots, L$. The jth LSH trie is built from the set of strings

$$\{(h_{1,j}(x), \ldots, h_{K,j}(x)) \mid x \in S\}. \tag{1}$$

where $h_{i,j} \sim \mathcal{H}$. The trie is constructed by recursively splitting 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.

Example 1. Figure 1 gives an example with a small set of points in the Euclidean plane, reporting the solution of a top-5 global join. The right hand side of the figure gives an example for an LSH forest with L tries initialized with the dataset. Each trie has depth K. Paths from the root to the leaves are labelled with the hash values of the corresponding points. For instance, in the first trie point d has hash value $(0, 1)$, while point f has hash value $(2, 2)$.

[3] It will be evident from the analysis of the algorithms that an *estimate* on the collision probability function will be sufficient in practice.

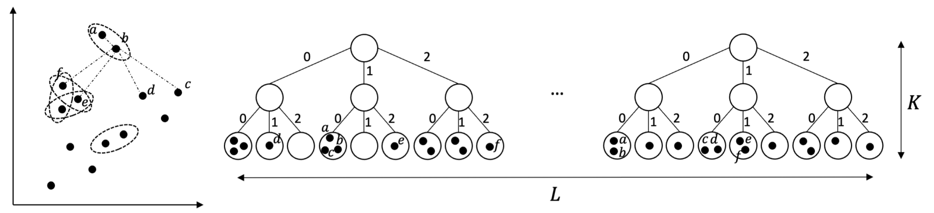


Fig. 1: Left: A set of 12 points in the plane with ellipses marking the 5 closest pairs. Right: L LSH tries of depth K with example distribution of points.

Given a query point $q \in \mathcal{X}$ and a failure probability δ, `PUFFINN` traverses each trie j to find the leaf corresponding to the string $(h_{1,j}(q), \ldots, h_{K,j}(q))$. Starting from that, it traverses the tries in a bottom-up fashion and keeps track of the current kth closest point x'_k. Let p be the probability of a collision under random choice of the LSH of two points at distance $\text{dist}(q, x'_k)$. If the current depth in the tries is i, and $\ln(1/\delta)/p^i$ is smaller than the current index of the trie that is inspected, the algorithm terminates and returns the closest k points as the answer to the query. [4, Lemma 3] shows that the stopping criterion guarantees that each point of the k nearest neighbors of q is found with probability at least $1-\delta$. Their Lemma 4 states that the algorithm asymptotically inspects $O(\text{OPT})$ candidate points in expectation, where OPT is the expected number of candidate points of a "clairvoyant" LSH-based algorithm that knows the distance distribution.

3 Algorithm, Analysis, and Problem Difficulty

In the following, we describe our algorithm to compute k closest pairs, extending the single query algorithm proposed in [4]. We first introduce some general concepts.

The algorithm makes use of a priority queue to keep track of the current k closest candidate pairs. This priority queue is implemented as a MaxHeap which associates a pair of points $(x, y) \in \mathcal{X}^2$ with their distance $\text{dist}(x, y)$. The number of elements in each priority queue will be at most k, i.e., if the priority queue has k elements and we insert an element, the element with maximum priority is removed. To make this assumption explicit, we call it a k-priority queue. Each operation in such a priority queue can be implemented to run in time $O(\log k)$, for example using a binary heap.

Fix a set $S \subseteq \mathcal{X}$ and two parameters $L, K \geq 1$. First we build L LSH tries of depth at most K as discussed in the previous section. For $x, y \in S$, let $x \sim_{i,j} y$ if $(h_{1,j}(x), \ldots, h_{i,j}(x)) \in T^i$ equals $(h_{1,j}(y), \ldots, h_{i,j}(y)) \in T^i$, i.e., x and y have the same length-i prefix in the jth LSH trie. Let $S_{i,j}$ denote the partition of points in S under the equivalence relation $\sim_{i,j}$. Since a trie can be built in linear

Algorithm 1: CLOSEST-PAIRS(k, δ)

```
PQ ← empty $k$-priority queue of unique (pair of points, dist)
for $i \leftarrow K, K-1, \ldots, 0$ do
    for $j \leftarrow 1, 2, \ldots, L$ do
        foreach $F \in S_{i,j}$ do
            foreach unchecked $(x, y) \in F$ do
                if $PQ.max() \geq \text{dist}(x, y)$ then
                    PQ.insert($(x, y)$, dist$(x, y)$)
        if $i == 0 \vee (\text{PQ.size()} == k \wedge j \geq \ln(1/\delta)/p(PQ.max())^i)$ then
            return PQ
```

time in the concatenated length of the input strings, cf. (1), we summarize the properties of building the trie data structure as follows.[4]

Fact 1 *Given K and L, building an LSH trie for n keys carries out $O(nKL)$ hash function evaluations to build the input strings, and takes time $O(nKL)$ and uses $O(nL)$ words of space to build the L tries representing these strings.*

Algorithm 1 describes the k-closest pair algorithm on a set $S \subseteq \mathcal{X}$ carried out on an LSH trie data structure with parameters $K, L \geq 1$. Given $k \geq 1$ and $\delta \in (0, 1)$, the algorithm initializes an empty k-priority queue. Using this data structure, the algorithm keeps track of the k closest pairs of points so far. We assume that S admits a total order (e.g., the indices of the keys in the set) and identify two points $x, y \in \mathcal{X}$ as the pair (x, y) with $x < y$.

For $i \leftarrow K$, the algorithm carries out all-to-all comparisons in each of the leaves, over all L tries. Next, the trie is traversed in a bottom-up fashion. For each node n on level i in trie j, i.e., the node that represents one set F in $S_{i,j}$, we carry out an all-to-all comparison between those $(x, y) \in F$ that have not been in the same subset in $S_{i+1,j}$. After trie j has been explored, we check the stopping criterion.

Example 2. Restricting Figure 1 to the first trie presented there, the algorithm carries out $3 + 3 + 1 + 1 = 8$ distance computation in the leaves of that trie. On the level higher up, it carries out $3 + 3 + 6 + 2 = 14$ distance comparisons. We observe that $S_{2,0}$ (the first trie at largest depth) has points $\{a, b, c\}$ and $\{e\}$ as part of the partition, while $S_{1,0}$ has the set $\{a, b, c, e\}$ and $S_{0,0}$ contains all twelve points in a single set.

To implement Algorithm 1, the leaves, i.e., all sets in $S_{K,j}$, are stored as sets using hashing. When traversing the trie, all-to-all comparisons are carried out between all pairs of child nodes and the sets are merged together. In this way, each

[4] Note here that the keys are the hash codes of the points in S. In many cases, a hash function can be evaluated in time $O(d)$, but many other scenarios exist. For example for set similarity, the MinHash value can be computed independently of the universe size in the size of the set at hand.

distance computation gives rise to two potential priority queue operations taking $O(\log k)$ time. We charge the cost of merging the child nodes to these all-to-all comparisons. Over all iterations of the nested loop, checking the termination criterion takes time $O(KL)$. We summarize this discussion in the following corollary. We separate the potential expensive distance computations from all other operations to make the statements more precise.

Corollary 1. *Let $S \subseteq \mathcal{X}$ with $|S| = n$, and let $K, L, k \geq 1$. Let C be the number of pairs (x, y) for which a distance is computed in Line 6 of Algorithm 1 in an LSH forest of depth K with L tries. The algorithm can be implemented to carry out $O(C)$ distance computations and all other operations run in time $O(C \log k + KL)$.*

3.1 Analysis

For a dataset $S \subseteq \mathcal{X}$, identify by the sequence $\mathrm{OPT} = ((x_1, y_1), \ldots, (x_k, y_k))$ a sequence of k closest pairs, and denote the best candidates found by Algorithm 1 as $\mathrm{OUT} = ((x'_1, y'_1), \ldots, (x'_k, y'_k))$.

Theorem 1. *Given $S \subseteq \mathcal{X}$, $k \geq 1$, and $\delta > 0$. Then*

$$\Pr[(x, y) \in OUT] \geq 1 - \delta \quad \forall (x, y) \in OPT.$$

Proof. Fix a pair (x, y) that is part of the output. There are two ways that the algorithm can return in Line 9. First, it can reach level 0, which means that it carried out a linear scan of all pairs of points. Second, it can return because it inspected the jth trie on level i and $j \geq \ln(1/\delta)/p(x'_k, y'_k)^i$. By the monotonicity of the LSH, $p(x'_k, y'_k) \leq p(x, y)$, because $\mathrm{dist}(x'_k, y'_k) \geq \mathrm{dist}(x, y)$. The probability that y did not collide with x in all j tries is

$$\begin{aligned}\left(1 - p(x, y)^i\right)^j &\leq \left(1 - p(x, y)^i\right)^{\ln(1/\delta)/p(x'_k, y'_k)^i} \\ &\leq \exp\left(-\ln(1/\delta) \cdot p(x, y)^i / p(x'_k, y'_k)^i\right) \\ &\leq \exp(-\ln(1/\delta)) \leq \delta,\end{aligned}$$

where the second-last inequality follows by the monotonicity of the LSH function, and we also used the inequality $1 - z \leq \exp(-z)$ for $z \geq 0$.

While Corollary 1 tells us that the running time of the algorithm is asymptotically equivalent to the number of pairs that are compared to each other, it is not clear how many such pairs will be inspected. To this end, let us define the work of an optimal, clairvoyant LSH-based closest pair algorithm that knows the distance distribution between all pairs of points. Let (x_k, y_k) be a k closest pair of points of maximum distance. We define

$$\mathrm{OPT}(L, K, k, \delta) = \min\left\{ \frac{\ln(1/\delta)}{p(x_k, y_k)^i} \left(i + \sum_{x, y \in S} p(x, y)^i \right) \,\middle|\, 0 \leq i \leq K, \frac{\ln(1/\delta)}{p(x_k, y_k)^i} \leq L \right\} \tag{2}$$

as the expected cost of the LSH-based algorithm that knows the exact distance distribution. The cost on level i includes that each pair of points in the top-k set is found with probability at least $1-\delta$ if we inspect $j \geq \ln(1/\delta)/p(x_k, y_k)^i$ tries. The expected cost of searching one LSH trie at depth i is $i + \sum_{x,y \in S} p(x,y)^i$. In our expression for the expected query time we use a unit cost model that counts distance computations. As shown in Corollary 1, counting distance computations is asymptotically equivalent to the running time of the algorithm.

The following theorem relates the running time of Algorithm 1 to the running time of the optimal algorithm that knows the full distance distribution.

Theorem 2. *Given a dataset S and parameters L, K, build the LSH trie data structure for S. Given k and δ such that $\ln(k/\delta) \leq L$, with probability at least $1-\delta$, Algorithm 1 computes the k closest pairs in S in expected time*

$$O\left(OPT(L, K, k, \delta/k) + L(k+K) + nKL\right).$$

Before proceeding with the proof, we remark that the analysis compares the expected time to the clairvoyant variant in the case that we set the failure probability so low that the result is *exact* (with probability at least $1-\delta$).

Proof. From Fact 1, building the trie takes time $O(nKL)$. Setting the failure probability to δ/k, Algorithm 1 returns the exact k closest pairs with probability at least $1-\delta$ using a union bound. Conditioning on this event, the algorithm will stop at the largest i such that $\frac{\ln(k/\delta)}{p(x_k,y_k)^i} \leq L$. Denote this level with i', and let i^* be the level used to minimize the work in (2). The expected running time of Algorithm 1 can be bounded by the term

$$\frac{\ln(k/\delta)}{p(x_k, y_k)^{i'}} \left(i' + \sum_{x,y \in S} p(x,y)^{i'} \right) + \left(L - \frac{\ln(k/\delta)}{p(x_k, y_k)^{i'}} \right) \left(i' + 1 + \sum_{x,y \in S} p(x,y)^{i'+1} \right), \tag{3}$$

where the first term bounds the work done on level i', and the second term bounds the work done on level $i'+1$ on the tries not inspected on the level above. Let T contain all pairs in S^2 that are not k closest pairs. We start by bounding the first term of the summation and continue as follows:

$$\begin{aligned}\frac{\ln(k/\delta)}{p(x_k, y_k)^{i'}} \left(i' + \sum_{x,y \in S} p(x,y)^{i'} \right) &\leq L(k+K) + \frac{\ln(k/\delta)}{p(x_k, y_k)^{i'}} \sum_{(x,y) \in T} p(x,y)^{i'} \\ &\overset{\text{(i)}}{\leq} L(k+K) + \frac{\ln(k/\delta)}{p(x_k, y_k)^{i^*}} \sum_{(x,y) \in T} p(x,y)^{i^*} \\ &\leq L(k+K) + \mathrm{OPT}(L, K, k, \delta/k),\end{aligned}$$

where (i) follows from the monotonicity of the LSH collision probability function. The theorem follows by observing that the second summand in (3) is at most a factor of $1/p(x_k, y_k) = O(1)$ larger than the first term.

We remark that the running time of Theorem 1 can be as high as $O(n^2)$ in the worst case. This is true because of the general nature of the problem (for example, by setting $k = \binom{n}{2}$), or because of the hardness of the data distribution.

3.2 Measuring the Difficulty of Closest Pairs

Ahle et al. [1] defined the expansion around the query as a difficulty measure to bound the running time of an LSH-based adaptive query algorithm. Aumüller and Ceccarello gave empirical evidence in [3] that the expansion predicts the indexing difficulty of datasets well in general. For the closest pair problem, we consider the following definition:

Definition 3. *Given $S \subseteq \mathcal{X}$ and $k, k' \geq 1$ with $k < k'$, let $((x_i, y_i))_{(x_i,y_i)\in S^2}$ be a sequence of pairs $(x, y) \in S^2$ ordered by their distance. Then contrast$_{k|k'}(S) := \frac{\mathrm{dist}(x_{k'}, y_{k'})}{\mathrm{dist}(x_k, y_k)}$ is the* contrast *of the kth to the k'th closest pair.*

We use this definition of contrast to bound the running time of the optimal, *clairvoyant* algorithm. By Theorem 1, this also provides a bound on the expected running time of Algorithm 1.

Lemma 1. *Given $S \subseteq \mathcal{X}$ with $|S| = n$, an LSH family $\mathcal{H}$, integers $K, L, k \geq 1$, and $\delta > 0$, let $c^* = contrast_{k|2k}(S)$. Let p_1, p_2 be the collision probability of the k and $2k$ closest pair, respectively, for $\mathcal{H}$. Let $\rho = \rho(c^*) = \frac{\log(1/p_1)}{\log(1/p_2)}$ and assume that $L = \Omega\left(n^{2\rho}/k^{\rho}\right)$. Then $OPT(K, L, k, \delta) = O\left(n^{2\rho}k^{1-\rho}\ln(1/\delta)\right)$.*

Proof. Let S' be the set of all pairs that are not among the $2k$ closest pairs. As discussed before, the expected cost on level i of the clairvoyant algorithm is

$$\frac{\ln(1/\delta)}{p_1^i}\left(i + \sum_{x,y\in S} p(x,y)^i\right) \leq \frac{\ln(1/\delta)}{p_1^i}\left(i + 2k + \sum_{(x,y)\in S'} p(x,y)^i\right)$$
$$\leq \frac{\ln(1/\delta)}{p_1^i}\left(i + 2k + \binom{n}{2}p_2^i\right).$$

Setting $i = \frac{\log(n^2/k)}{\log(1/p_2)}$, $\binom{n}{2}p_2^i \leq k$ and $1/p_1^i = (n^2/k)^{\frac{\log 1/p_1}{\log 1/p_2}} = n^{2\rho}/k^{\rho}$.

For Euclidean space, $\rho(c) = 1/c^2$, so their exists a level for the clairvoyant algorithm with subquadratic expected running time $O(n^{2/c^2}k^{1-1/c^2}\ln(1/\delta))$. As shown in Theorem 2, Algorithm 1 has the same asymptotic running time up to logarithmic factors. Note that $c \geq \sqrt{2}$ yields *sublinear* running time, because the build time of the trie data structure is disregarded. The expected running time of Algorithm 1 is at least $\tilde{O}(nKL)$ for building the trie.

If the contrast is small, the space requirement on L in Lemma 1 is large. Let c^* be the smallest value of c such that $L \geq n^{2\rho(c^*)}/k^{\rho(c^*)}$, and let d^* be the distance of a k-th closest pair. We can carry out the same analysis as in the proof of Lemma 1. In each trie, we have expected cost k for the k

closest pairs, and we expect to see no more than $\binom{n}{2}p(c^*d^*)^i \leq k$ pairs at distance larger than c^*d^* for the choice $i = \lfloor \log(L)/\log(1/p_1) \rfloor$. Each pair with a distance in $[d^*, c^*d^*]$ collides with probability at most p_1^i, so overall all tries we expect to see each pair once. Thus, the expected number of pairs inspected is $O\left((Lk + N_{c^*,d^*}(S))\ln(1/\delta)\right) = O\left(\left(n^{2\rho(c^*)}k^{1-\rho(c^*)} + N_{c^*,d^*}(S)\right)\ln(1/\delta)\right)$, where $N_{c^*,d^*}(S)$ is the number of pairs with distance in $[d^*, c^*d^*]$.

4 Implementation Choices

Our algorithms are implemented in the framework provided by PUFFINN [4].

Trie data structure. We focus on supporting the Cosine and the Jaccard similarity. For these two similarity functions we choose as hash functions 1-bit MinHash [20] and SimHash [9], respectively. Both these hash functions output single bits: it is thus very natural to represent the strings of hash values described in Section 2 as bitstrings, packing the bits into machine words. We also support more complex hash functions such as Crosspolytope-LSH [2]. For this LSH family, we view the output hash code (which is an integer $\{0, \ldots, 2d-1\}$) as a length-$\lceil \log d + 1 \rceil$ bitstring and concatenate a small number of hash functions. For intermediate positions in the trie, i.e., those where we use only part of the output of a single LSH, we estimate the collision probabilities by sampling. Since evaluating $O(nKL)$ hash values is time-demanding, PUFFINN supports the tensoring and pooling approach described by Christiani in [10].

By viewing the output of the LSH as a bitstring, we can optimize the trie implementation. Rather than using a pointer-based implementation, we store point indices, paired with the corresponding bitstring hash values, in a flat array. The array is then sorted lexicographically by hash value, leading to a more compact and cache efficient data structure. Furthermore, to speed up index construction we rely on radix-sorting, given that the bitstring hash values can be also interpreted as integers. In this implementation of the trie, the nodes in the same subtrie at a given depth i are all the consecutive entries of the array sharing the same length-i prefix.

Sketching Finally, to further prune similarity computations we use sketching with a similar setup as the original PUFFINN paper [4]. Each point is associated with a different 64-bit sketch in each repetition, computed using either 1-bit MinHash or SimHash, depending on the similarity function. Consider now two colliding points x and y, and let s_k be the highest distance of any pair currently in the k-heap to be possibly updated, if $d(x, y) < s_k$. Let τ be the expected number of different bits in the sketches of points at distance s_k. Before evaluating the similarity of x and y, we first check the number of different bits in the corresponding sketches: if such difference is larger than τ, the similarity between x and y is not computed at all. This has the effect of reducing the number of similarities being evaluated, at the cost of slightly reducing the recall of the algorithm.

In the following, we refer to the implementation of our algorithms as `PUFFINN-join`.

dataset	n	dimensions	RC @ 100	RC @ 10 000	c. @ 100	c. @ 10 000
DeepImage	10 000 000	96	7 615.56	2 343.25	1.25	1.31
Glove	1 193 514	200	38.04	5.15	1.33	1.29
DBLP	2 773 660	4 405 478	22.52	7.83	1.15	1.33
Orkut	2 732 271	8 730 857	20.97	2.99	1.39	1.36

Table 1: Datasets used in the experimental evaluation. The last two columns report the relative contrast at 100 pairs and 10 000 pairs [17].

5 Evaluation

This section reports on the results of our experiments, which are tailored to answer the following questions: (Q1) How does our approach compare with the state of the art? (Q2) How does the amount of available memory influence the performance of our algorithm? (Q3) What is the relationship between intrinsic dimensionality measures and the performance of the algorithm?

Experimental Setup. Experiments were run on 2x 14-core Intel Xeon E5-2690v4 (2.60GHz) with 512GB RAM using Ubuntu 16.10 (kernel 4.4.0). The code is available at `https://github.com/Cecca/puffinn`, along with all the scripts to suitably preprocess the datasets.

We focus our evaluation on two metrics: the running time and the recall.

Datasets. Information about the datasets used in this evaluation is reported in Table 1. In particular, we consider two datasets with cosine similarity (`Glove` and `DeepImage`) and two datasets under Jaccard similarity (`DBLP` and `Orkut`).

In particular, for all datasets we report a summary of the Relative Contrast [17] — i.e. the ratio between the average distance and the k-th distance — which will be useful in interpreting the results [3]. In particular, we expect `DeepImage` to be easier than `Glove`, and both to be easier than the two Jaccard datasets. Furthermore, the relative contrast of `DeepImage` is extremely high. This means that the top pairs of globally closest points are much closer than the average pair, meaning that this dataset is expected to be considerably easier than the others for the global top-k problem.

Baselines. Under the Jaccard similarity we compare with `XiaoEtAl` [26], whereas for the cosine similarity[5] we consider the `LSB-Tree` approach [24, Algorithm CP3] Furthermore, for cosine similarity we consider a baseline that uses the HNSW implementation provided by FAISS [19], querying the k-nearest neighbors of each point and then selecting the k closest among the resulting pairs.

Parameter choices. For our approach we set the memory given to the index in the range 256MB to 32GB, by powers of two, corresponding to up to $L \approx 2000$, depending on the dataset, for fixed $K = 24$. As for the target recall, we set it to

[5] We omit PM-LSH [28] as its closest-pair implementation is unavailable and as we were unsuccessful in both implementing it ourselves and in reaching out to the authors.

Table 2: Running times. Missing values are for runs that timed out after 8 hours. The last column reports the time for the index construction (not applicable to `XiaoEtAl`), which is also included in the total time reported in the other columns

		Total time (s) for different k					indexing (s)
dataset	algorithm	1	10	100	1 000	10 000	
Glove	faiss-HNSW	68.1	132.8	551.7	-	-	63.8
	LSBTree	18.2	136.7	2028.4	2127.4	959.3	3.1
	PUFFINN	5.0	5.0	5.0	5.1	6.3	4.7
DeepImage	faiss-HNSW	299.7	533.8	2632.9	-	-	255.4
	LSBTree	112.0	93.4	114.6	176.2	368.6	13.6
	PUFFINN	37.2	37.5	37.1	37.4	37.4	18.9
DBLP	XiaoEtAl	9.3	14.0	9.8	12.1	58.3	0.0
	PUFFINN	4.9	4.9	4.9	4.9	5.0	4.2
Orkut	XiaoEtAl	118.0	122.0	142.3	1170.3	-	0.0
	PUFFINN	24.7	24.8	24.7	24.5	73.3	23.9

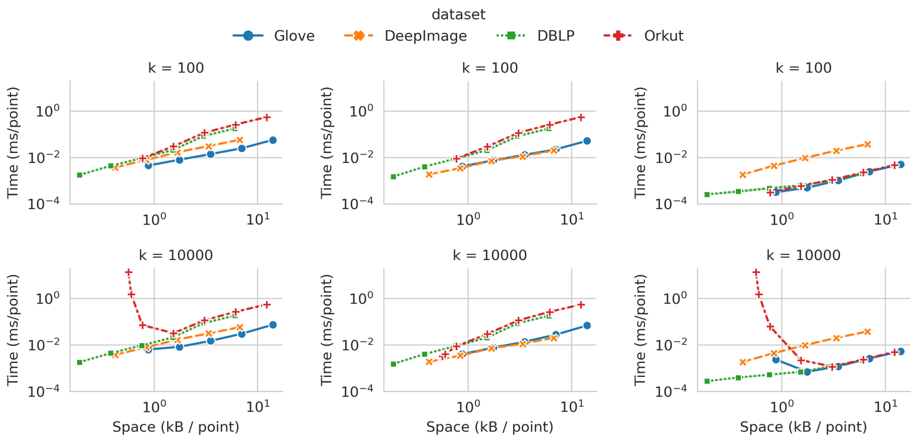
0.8, 0.9, and 0.99. For `HNSW` we test $M \in [32, 48]$, $efConstruction \in [100, 500]$, $efSearch \in [k8, k16]$ for a top-k join. For `LSB-Tree` we test m up to 8, whereas `XiaoEtAl` takes no parameters. We remark that this is one of the most relevant differences between our approach and the state of the art: while we can specify a desired target recall, all other approaches require to experiment with several combinations of parameters before finding a configuration suitable for the desired quality level.

Comparison with baselines. In the first set of experiments we measure the running time required by different algorithms to achieve recall at least 0.9. The results are reported in Table 2. On all datasets, `PUFFINN-join` runs faster than the baselines. Furthermore, observe that for $k \leq 1000$ the running time of our algorithm remains basically unaffected by the number of pairs returned. This is because all of the runtime, in this setup, is spent building the index, which is independent of the value of k. Finally, observe that compared to `LSBTree`, our approach is orders of magnitude faster.

Space/time tradeoffs. In Figure 2 we report the space/time tradeoff of our algorithm, at recall 0.9 and $k \in \{100, 10\,000\}$. In particular, we report on the total time (Figure 2a), which is comprised of the time to build the index (Figure 2b), and the time to run the join (Figure 2c). The top row of plots reports the results for $k = 100$, the bottom row for $k = 10\,000$.

Observe that the indexing time (Figure 2b) increases with the space given to the index, as expected. Furthermore, we have the Jaccard-similarity datasets being processed more slowly than the cosine-similarity datasets: this is a consequence of the longer time required to compute MinHash compared to SimHash.

For $k = 100$, on all datasets, the best performance is attained by the configurations using the least space. The reason is in the high relative contrast values of the 100th top pair (see Table 1), which imply that in the few repetitions required



(a) Total time (b) Indexing time (c) Join time

Fig. 2: Space/time tradeoff of our algorithm at guaranteed recall 0.9 and $k \in \{100, 10\,000\}$, for the global top-k problem.

to confirm the top pairs there are few other collisions to check. In fact, the index construction time dominates the join time (Figure 2c) by a large margin.

In contrast with this for $k = 10\,000$ we have that for `Orkut` increasing the memory usage gives better performance. In fact, the 10 000th pair of this dataset has relative contrast of just about 3, meaning that our approach requires either many repetitions or short hash values to confirm it: using more memory allows to use more repetitions on longer hashes, thus reducing the number of collisions to be evaluated. Using too little memory makes the join part of the algorithm dominate on the index construction part, enabling the tradeoff that is observed in the figure.

References

1. Ahle, T.D., Aumüller, M., Pagh, R.: Parameter-free locality sensitive hashing for spherical range reporting. In: SODA. pp. 239–256. SIAM (2017)
2. Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I.P., Schmidt, L.: Practical and optimal LSH for angular distance. In: NIPS. pp. 1225–1233 (2015)
3. Aumüller, M., Ceccarello, M.: The role of local dimensionality measures in benchmarking nearest neighbor search. Inf. Syst. **101**, 101807 (2021)
4. Aumüller, M., Christiani, T., Pagh, R., Vesterli, M.: PUFFINN: parameterless and universally fast finding of nearest neighbors. In: ESA. LIPIcs, vol. 144, pp. 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
5. Bawa, M., Condie, T., Ganesan, P.: LSH forest: self-tuning indexes for similarity search. In: WWW. pp. 651–660. ACM (2005)
6. Broder, A.Z.: On the resemblance and containment of documents. In: Compression and Complexity of Sequences 1997. Proceedings. pp. 21–29. IEEE (1997)

7. Cai, X., Rajasekaran, S., Zhang, F.: Efficient approximate algorithms for the closest pair problem in high dimensional spaces. In: PAKDD (3). LNCS, vol. 10939, pp. 151–163. Springer (2018)
8. Chan, T.M.: Orthogonal range searching in moderate dimensions: k-d trees and range trees strike back. In: SoCG. LIPIcs, vol. 77, pp. 27:1–27:15 (2017)
9. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: STOC. pp. 380–388. ACM (2002)
10. Christiani, T.: Fast locality-sensitive hashing frameworks for approximate near neighbor search. In: SISAP. LNCS, vol. 11807, pp. 3–17. Springer (2019)
11. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB. pp. 426–435. Morgan Kaufmann (1997)
12. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SCG. pp. 253–262. ACM (2004)
13. Dietzfelbinger, M., Hagerup, T., Katajainen, J., Penttonen, M.: A reliable randomized algorithm for the closest-pair problem. J. Algorithms **25**(1), 19–51 (1997)
14. Dong, W., Wang, Z., Josephson, W., Charikar, M., Li, K.: Modeling LSH for performance tuning. In: CIKM. pp. 669–678. ACM (2008)
15. Gao, Y., Chen, L., Li, X., Yao, B., Chen, G.: Efficient k-closest pair queries in general metric spaces. VLDB J. **24**(3), 415–439 (2015)
16. Har-Peled, S., Indyk, P., Motwani, R.: Approximate nearest neighbor: Towards removing the curse of dimensionality. Theory Comput. **8**(1), 321–350 (2012)
17. He, J., Kumar, S., Chang, S.: On the difficulty of nearest neighbor search. In: ICML. icml.cc / Omnipress (2012)
18. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Proceedings of 30th Annual ACM Symposium on the Theory of Computing (STOC). pp. 604–613 (1998)
19. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with gpus. IEEE Trans. Big Data **7**(3), 535–547 (2021)
20. Li, P., König, C.: B-Bit Minwise Hashing. In: WWW. pp. 671–680. WWW '10, ACM, New York, NY, USA (2010)
21. Ntarmos, N., Patlakas, I., Triantafillou, P.: Rank join queries in nosql databases. Proc. VLDB Endow. **7**(7), 493–504 (2014)
22. Pirbonyeh, M., Rezaie, V., Parvin, H., Nejatian, S., Mehrabi, M.: A linear unsupervised transfer learning by preservation of cluster-and-neighborhood data organization. Pattern Anal. Appl. **22**(3), 1149–1160 (2019)
23. Skopal, T., Pokorný, J., Snásel, V.: Nearest neighbours search using the pm-tree. In: DASFAA. LNCS, vol. 3453, pp. 803–815. Springer (2005)
24. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. ACM Trans. Database Syst. **35**(3), 20:1–20:46 (2010)
25. Wang, H., Yang, L., Xiao, Y.: Setjoin: a novel top-k similarity join algorithm. Soft Comput. **24**(19), 14577–14592 (2020)
26. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k set similarity joins. In: ICDE. pp. 916–927. IEEE Computer Society (2009)
27. Yang, Z., Zheng, B., Li, G., Zhao, X., Zhou, X., Jensen, C.S.: Adaptive top-k overlap set similarity joins. In: ICDE. pp. 1081–1092. IEEE (2020)
28. Zheng, B., Zhao, X., Weng, L., Nguyen, Q.V.H., Liu, H., Jensen, C.S.: PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search. VLDB J. **31**(6), 1339–1363 (2022)

29. Zhou, X., Wu, B., Jin, Q.: Analysis of user network and correlation for community discovery based on topic-aware similarity and behavioral influence. IEEE Trans. Hum. Mach. Syst. **48**(6), 559–571 (2018)