Information Systems 51 (2015) 43–61

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Near neighbor searching with *K* nearest references

E. Chávez[a,1], M. Graff[c,1], G. Navarro[b,2], E.S. Téllez[c,*,1]

[a] CICESE, Mexico
[b] CeBiB – Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile
[c] INFOTEC/Cátedra CONACyT, Mexico

ARTICLE INFO

Article history:
Received 26 June 2014
Received in revised form
4 December 2014
Accepted 3 February 2015
Recommended by: F. Korn
Available online 7 February 2015

Keywords:
Proximity search
Searching by content in multimedia databases
k nearest neighbors
Indexing metric spaces

ABSTRACT

Proximity searching is the problem of retrieving, from a given database, those objects closest to a query. To avoid exhaustive searching, data structures called *indexes* are built on the database prior to serving queries. The *curse of dimensionality* is a well-known problem for indexes: in spaces with sufficiently concentrated distance histograms, no index outperforms an exhaustive scan of the database.

In recent years, a number of indexes for *approximate* proximity searching have been proposed. These are able to cope with the curse of dimensionality in exchange for returning an answer that might be slightly different from the correct one.

In this paper we show that many of those recent indexes can be understood as variants of a simple general model based on *K-nearest reference* signatures. A set of references is chosen from the database, and the signature of each object consists of the *K* references nearest to the object. At query time, the signature of the query is computed and the search examines only the objects whose signature is close enough to that of the query.

Many known and novel indexes are obtained by considering different ways to determine how much detail the signature records (e.g., just the set of nearest references, or also their proximity order to the object, or also their distances to the object, and so on), how the similarity between signatures is defined, and how the parameters are tuned. In addition, we introduce a space-efficient representation for those families of indexes, making it possible to search very large databases in main memory. Small indexes are cache friendly, inducing faster queries.

We perform exhaustive experiments comparing several known and new indexes that derive from our framework, evaluating their time performance, memory usage, and quality of approximation. The best indexes outperform the state of the art, offering an attractive balance between all these aspects, and turn out to be excellent choices in many scenarios. Our framework gives high flexibility to design new indexes.

1. Introduction

Proximity searching is the problem of retrieving objects close to a given query, from a possibly large collection (also called a database), under a (dis)similarity measure. It is an ubiquitous problem in computer science, with applications in machine learning, pattern recognition, statistics, bioinformatics, and textual and multimedia information retrieval, to name a few. Many solutions have been discussed

[*] Correspondences to: Circuito Tecnopolo Sur No. 112 Col. Fracc. Tecnopolo Pocitos, 20313, Aguascalientes, Ags. Tel.: +52 (55) 5624-2800x6353.
E-mail addresses: elchavez@cicese.mx (E. Chávez), mario.graff@infotec.com.mx (M. Graff), gnavarro@dcc.uchile.cl (G. Navarro), eric.tellez@infotec.com.mx (E.S. Téllez).

[1] Partially funded by (CONACyT Grant 179795) Mexico.

[2] Funded with basal funds FB0001, Conicyt, Chile.

http://dx.doi.org/10.1016/j.is.2015.02.001
0306-4379/© 2015 Elsevier Ltd. All rights reserved.

in application papers, surveys [10,16,28,50] and even books [47,57]. The trivial solution of comparing the query to each database object is acceptable only when the collection is small and the comparison is inexpensive. This sequential scan does not scale well to most realistic cases, where it is necessary to preprocess the database to answer queries more efficiently.

The most popular query in applications is that of retrieving the k *nearest neighbors* of a query, that is, the k database objects closest to it. In one dimension (i.e., objects are just numbers) the problem has a trivial solution in time $O(\log n+k)$ using balanced search trees, but the nature of the problem changes in more dimensions. In two dimensions, structures based on the Voronoi diagram still allow $O(\log n)-$time searching for the nearest neighbor [31]. This has been generalized to the k nearest neighbors in $O(\log n+k)$ time, yet the space grows to $O(kn)$ [33]. However, those methods do not generalize to higher dimensions. In δ dimensions, kd-trees [9] provide nearest-neighbor searches in $O(\delta n^{1-1/\delta}+k)$ time, for a database of n objects. As predicted by the complexity formula, their practical performance is acceptable in low dimensions, but already for dimension 5 or so they become slower than a sequential scan. This is not just a shorcoming of kd-trees, but a general phenomenon that affects every index, known as the *curse of dimensionality*.

In the case of strings, under Hamming or Levenshtein distances, some specific techniques can be applied [12]. A well-known one is to cut the query string into shorter ones that can be searched for with smaller radii (or even exactly). Such techniques turn out to be useful in specific areas like computer vision [40].

The curse of dimensionality arises also in general metric spaces, where the objects are not necessarily δ-dimensional points, but the distance function defined on them is a metric. The empirical evidence shows that, the more concentrated is the histogram of the distances, the harder to index is the space regardless of the index used. Those spaces are said to have high *intrinsic dimension*. This issue is well documented in the literature, for example by Pestov [43–46], Volnyansky and Pestov [55], Shaft and Ramakrishnan [48], Indyk [30], Samet [47], Hjaltason and Samet [28], Chavez et al. [16], Böhm et al. [10], and Skopal and Bustos [50].

It has been shown that the curse of dimensionality can be sidestepped by relaxing the conditions of the search problem, so as to allow the algorithm to return a “good approximation” to the actual answer. Then the quality of the approximation becomes an issue. One possible measure is the degree of coincidence between the true and the returned set of k nearest neighbors. Another is the ratio of the distances between the query and the true versus the returned kth nearest neighbor. For example, in δ-dimensional spaces it is possible to return an element within $1+\epsilon$ times the distance to the true nearest neighbor in time $O(\delta(1+6\delta/\epsilon)^{\delta}\log n)$ [4]. Note that the curse of dimensionality is still there, but it can be traded for the quality of the answer.

In the case of metric spaces, Chavez et al. [14] introduced a novel idea for an approximate search index based on *permutations*. A set of references was chosen from the database and each object was assigned a *signature* consisting of ordering the references by increasing distance to the object. Then a query was compared to the reference objects and its signature was computed. Proximity between the query and the objects was then predicted without directly measuring distances, but rather comparing the signatures. The index obtained excellent empirical results in real-life applications, and opened a new fruitful research area on this topic [2,53,18,54].

In this paper we generalize upon all those ideas, introducing a framework we call *K*-nearest references ($\mathsf{K}-\mathsf{nr}$). In $\mathsf{K}-\mathsf{nr}$, each object will be represented by a subset of a set of references, and the signature computed from that subset of references will be variable (e.g., it can consider or ignore the distance ordering to the object, the precise distances, etc.). Depending on these choices, on how we measure the distance between two signatures, and the parameter tuning, we obtain known and new indexes. We also introduce a specific data representation that allows one to store the index within little space, this way enabling maintaining larger databases in fast main memory. A number of specific algorithms can be supported by the same data structure by fixing parameters of the general framework, giving much flexibility to design and test new techniques. Several speed/recall/memory/preprocessing tradeoffs can be obtained, depending on the application.

We provide extensive experimental evidence showing how novel combinations derived from our approach surpass the state of the art in several aspects. We include in this comparison the best variants of Locality Sensitive Hashing (LSH) [24].

The rest of the paper is organized as follows: Section 2 formalizes the problem and recalls basic concepts and previous work. Section 3 describes the $\mathsf{K}-\mathsf{nr}$ framework and shows how it can be instantiated into all the existing reference based indexes. New indexes are also proposed by plugging other combinations into the framework. Section 4 describes an efficient implementation of $\mathsf{K}-\mathsf{nr}$ indexes. The experimental setup is detailed in Section 5, and a thorough set of experiments comparing the various aspects of $\mathsf{K}-\mathsf{nr}$ indexes is presented in Section 6. Section 7 compares $\mathsf{K}-\mathsf{nr}$ indexes with the best LSH variants we are aware of. We summarize our results in Section 8.

2. Preliminaries

A *metric space* is a tuple (U, d) where U is the domain and $d: U\times U\rightarrow\Re^{+}$ is a *distance*, that is, for all $u,v,w\in U$ it holds (1) $d(u,v)=0\iff u=v$, (2) $d(u,v)=d(v,u)$, and the *triangle inequality* (3) $d(u,w)+d(w,v)\geq d(u,v)$.

A *database* will be a finite set $S\subset U$ of size n, which can be preprocessed to build an *index*. We are interested in solving *k-nearest neighbor queries* with the index, that is, given a query $q\in U$, return a set $\mathsf{k}-\mathsf{nn}(q)$ of k elements of S closest to q. Typical values of k in applications is a few tens.

In this paper we focus primarily on *approximate* k-nn searches, where we are allowed to return a set $\mathsf{ak}-\mathsf{nn}(q)$ slightly different from $\mathsf{k}-\mathsf{nn}(q)$. We will use *recall* to measure the quality of approximation:

$$\text{recall}=\frac{|\mathsf{ak}-\mathsf{nn}(q)\cap\mathsf{k}-\mathsf{nn}(q)|}{k},$$

that is, the proportion of true answers found. Note that, since $|\mathsf{k}-\mathsf{nn}(q)|=|\mathsf{ak}-\mathsf{nn}(q)|=k$, this concept coincides with both recall and precision measures in information retrieval [6]. Another measure is the *proximity ratio*, $(\max_{u \in \mathrm{ak-nn}(q)} d(u,q))/(\max_{u \in \mathrm{k-nn}(q)} d(u,q))$, that is, how much farther from q is the kth object found compared to the true kth nearest neighbor. However, this is not a good measure in high-dimensional metric spaces, since all the distances are similar and thus even a random answer can yield, on average, proximity ratios close to 1 (we have verified this phenomenon in our experimental setup as well).

2.1. Example distance functions

When the metric space is the δ-dimensional vector space, Minkowski's L_p distances are commonly used:

$$L_p(u,v) = \left(\sum_{i=1}^{\delta} |u_i - v_i|^p \right)^{1/p},$$

where u_i/v_i are the ith components of the vectors u/v. An L_p evaluation requires $O(\delta)$ basic operations. Three instances are used most often: the L_1 distance (called *Manhattan* distance), the L_2 distance (called *Euclidean*), and the *Maximum* distance $L_\infty = \max_{1 \le i \le \delta} |u_i - v_i|$.

In information retrieval, *tf-idf vectors* are used to represent documents [6], and the proximity is measured with the *cosine similarity*,

$$\cos(u,v) = \frac{\sum_{i=1}^{\delta} u_i v_i}{\sqrt{\sum_{i=1}^{\delta} u_i^2} \cdot \sqrt{\sum_{i=1}^{\delta} v_i^2}}.$$

The angle between vectors is a distance, $d_C(u,v) = \arccos \mathrm{sim}_C$.

The *Hamming distance* is applied to strings of fixed length ℓ. The strings are aligned and we count the mismatches, that is,

$$d_H(u,v) = \sum_{i=1}^{\ell} \begin{cases} 0 & \text{if } u_i = v_i, \\ 1 & \text{if } u_i \neq v_i. \end{cases} \tag{1}$$

Another interesting case is that of metric spaces of sets. A commonly used distance on sets is one minus Jaccard's coefficient, that is,

$$d_J(u,v) = 1 - \frac{|u \cap v|}{|u \cup v|}.$$

2.2. Approximate proximity searching

The curse of dimensionality has motivated research on approximate indexes. There exist generic techniques to convert a given exact index into approximate, by essentially pruning the search process when a time budget is exceeded, and prioritizing the progress of the search by exploring first the parts of the dataset that are estimated to be more promising. These techniques may offer just empirical or more formal guarantees, and the latter can be of the form of an exact or probabilistic guarantee on the quality of the results. There is a large number of such approaches in the literature; see Patella and Ciaccia [42] for an exhaustive review.

Skopal [49] introduced an alternative model unifying both exact and approximate proximity search algorithms. He also introduced the TriGen algorithm, a method to convert a (dis)similarity function into a metric function with high probability. This technique allows indexing general similarity spaces using metric indexes.

Kyselak et al. [32] observed that most approximate methods optimize the average accuracy of the indexes, but it is common to find very bad performances on individual queries. They propose a simple solution to stabilize the accuracy. The idea is to reduce the probability of a poor-quality result by using multiple independent indexes that solve the query in parallel, so that with high probability at least one index achieves good quality on average for every query.

Even with these techniques, the search cost is generally too high for practical applications. A new family of techniques designed to be approximate from the beginning started with the Permutation Index (PI) [14] and now include the Brief Permutation Index (BPI) [53], the Metric Inverted File (MIF) [2], the PP-Index [18], and the Compressed Neighborhood Approximation Inverted Index (CNAPP) [54]. All these techniques can be described from a common perspective, which is the focus of this paper. We defer their discussion to Section 3.1.

2.3. Locality sensitive hashing

Gionis et al. [24] introduced Locality Sensitive Hashing (LSH), a fast approximate proximity searching technique giving probabilistic guarantees on the quality of the result. The general idea of an LSH index is to find hashing functions such that close items share the same bucket with high probability, while distant items share the same bucket with low probability.

More formally, a family of hashing functions $\mathcal{H} = \{g_1, g_2, \ldots, g_h\}$, $g_i : U \to \{0,1\}$ is called (p_1, p_2, r_1, r_2)−sensitive if, for any $p, q \in U$,

- if $d(p,q) < r_1$ then $Pr[\mathrm{hash}(p) = \mathrm{hash}(q)] > p_1$,
- if $d(p,q) > r_2$ then $Pr[\mathrm{hash}(p) = \mathrm{hash}(q)] < p_2$,

where $\mathrm{hash}(u)$ is the concatenation of the output of individual hashing functions g_i, following a fixed order, that is, $\mathrm{hash}(u) = g_1(u) g_2(u) \ldots g_h(u)$.

Notice that LSH gives guarantees in terms of distances, not in terms of recall. To achieve the desired p_1 and p_2 values, many empty buckets may have to be created (for potential queries of U that are far from any element in S). This poses a memory problem, which can be alleviated by combining several hashing function families. Still, high-quality indexes usually need high amounts of memory.

LSH can be defined for general metric spaces, but the process of finding suitable hashing functions g_i is not trivial and has to be engineered for each case. There exist simple LSH functions the Euclidean (L_2), Hamming, Jaccard, and Cosine distances [3,24].

Data Sensitive Hashing (DSH) [23] is an LSH-based index, where hashing functions are selected and weighted from a pool of hashes. The idea is to generate a strong classifier based on several weak classifiers (hashing functions) using the Boosting algorithm [22]. This methodology is in contrast with the traditional uniform partitioning of the space used by other LSH techniques. Notice that DSH is independent of the actual hashing functions, thus any LSH function can be used.

DSH indexes are constructed to solve k nearest neighbor queries, thus they fit better than LSH in our scenario. Gao et al. [23] compare DSH with traditional LSH and other learning based methods, and show that DSH surpasses most of them in memory, recall and speed.

3. The $\mathsf{K-nr}$ framework

A Voronoi diagram [5] is built on a set R of two-dimensional points. It partitions the plane into $|R|$ regions. Each region is formed by the points closest to one of the elements in R. The generalization of Voronoi diagrams to general metric spaces is called a *Dirichlet partition*, and there are exact and approximate metric indexes based on it [13,17,38,34,35]. In the Dirichlet partition of a set of *references* $R \subset S$, each element of S is assigned to the partition corresponding to the closest reference.

Lee [33] defined *Kth order Voronoi diagrams*, where the plane is partitioned not only according to which element of R is closest to the point, but also which is the second, third, …, Kth closest. We define analogously the *Kth order Dirichlet partition* in a general metric space.

Definition 3.1. Given a metric space on universe U, and a finite database $S \subset U$, the *Kth order Dirichlet partition* induced by a set of *references* $R \subset U$ of size $|R| = \rho$ is a function $\mathsf{K-nn}: U \to R^K$ that assigns to each $u \in S$ the tuple $\hat{u} = \mathsf{K-nn}(u) = \langle r_1, \ldots, r_K \rangle$, such that r_i is the ith closest element to u in R. The tuple $\hat{u}$ is called the *signature* of u.

The key idea of the $\mathsf{K-nr}$ framework is to use the similarity between $\hat{u}$ and $\hat{q}$ as an estimation of the closeness between u and q (see Fig. 1). Formally, we can define a $\mathsf{K-nr}$ index as follows.

Definition 3.2. A $\mathsf{K-nr}$ index is a tuple $(U, d, S, R, K, \mathsf{K-nn}, \mathsf{sim})$, where (U, d) is a metric space, $\mathsf{K-nn}$ is a Kth order Dirichlet domain on the references R of the database S, and $\mathsf{sim}: R^K \times R^K \to \Re^+$ is a similarity function between signatures $\mathsf{K-nn}(\cdot)$.

In the general spirit of examining a subset of promising elements of S to solve an approximate proximity query, we will fix a parameter γ and examine the γ elements $u \in S$ whose signatures are most similar to that of q. More precisely, the search follows these steps:

1. We compute the signature $\hat{q} = \mathsf{K-nn}(q)$.
2. We find the γ objects $u \in S$ with maximum $\mathsf{sim}(\hat{u}, \hat{q})$.
3. We evaluate $d(q, u)$ for all those γ candidates and return the k closest ones to q.

Therefore, the total number of distance computations is $\rho + \gamma$, from steps 1 and 3, respectively. The major extra CPU work is related to how we find the γ objects whose signature is closest to $\hat{q}$, in step 2.

This general idea can be instantiated in several ways. The most important decision is how we define sim. The better its ability to hint the actual closeness, the more accurate will be the method for a given amount of work.

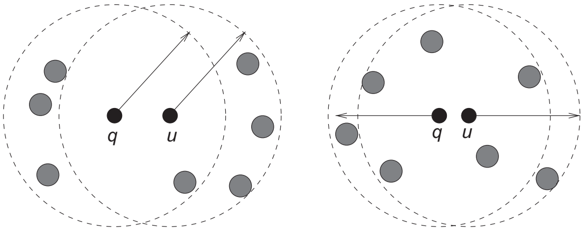

Fig. 1. Shared references as proximity predictors. Small black balls are objects in S and big gray balls are references. A bad case is shown on the left, and a good (more likely one) on the right.

The way signatures are compared also dictates how much information must be stored on them, which influences the space usage of the index. The complexity of computing sim also influences the CPU cost of the algorithm. The other decisions are the tuning of the parameters γ, ρ, and K.

- A larger value of γ increases the time cost, but improves the quality of the approximation.
- The size of R, ρ, can be between K and n. The larger it is, the costlier it is to compute $\mathsf{K-nn}(q)$, but the K references of each object can be chosen from a more representative set. Note, however, that if $\rho$is too large compared to K, then even the signatures of close objects may be different.
- The K value affects the space usage of the index. Generally, a larger K value improves the quality of the approximation as it gives a finer-grained signature. However, this depends on the way the similarity between signatures is computed. Too fine-grained signatures may introduce noise and distort the significance of similarity functions. As an obvious example, if sim is simply the number of common references in both tuples, regardless of their position in the tuple, and we use $K = \rho$, then all the signatures will be equal. A larger K also generally increases the CPU time needed to find the γ candidates.

For simplicity we will relabel the elements of R as $\{1, \ldots, \rho\}$. Next we show how several existing indexes can be regarded as particular cases of the $\mathsf{K-nr}$ framework, and later introduce new ones.

3.1. *Existing proximity indexes as* $\mathsf{K-nr}$ *instances*

Our framework captures the behavior of at least five indexes found in the literature: Permutation Index [14], Brief Permutation Index [53], Metric Inverted File [2,1], Prefix Permutations Index [18,19], and Succinct Nearest Neighbor Search [54]. Below we describe them within the $\mathsf{K-nr}$ framework.

3.1.1. *Permutation Index (PI)*

Chavez et al. [14] introduced the idea of choosing a set of K database elements called the *permutants* and associating to each $u \in S$ a signature formed by the permutants sorted by increasing distance to u. That is, they associate to each u a permutation Π_u of $[1..K]$. This fits in our framework if we consider the permutants as the set R of $\rho = K$

references, so that all the references are chosen in the signature, and thus $\mathsf{K}-\mathsf{nn}(u)=\Pi_u$.

They explored various similarity measures between permutations, including Spearman Footrule and Spearman Rho. Spearman Foortrule is defined as the sum of the distances between the positions of the same permutant in both permutations, and Spearman Rho as the sum of the squared distances. More formally, let Π^{-1} denote the inverse permutation of Π, then

$$\mathsf{sim}_{\mathsf{PI-Foortule}}(\hat{u},\hat{v})=K^2-L_1(\Pi_u^{-1},\Pi_v^{-1}),$$
$$\mathsf{sim}_{\mathsf{PI-Rho}}(\hat{u},\hat{v})=K^3-L_2^2(\Pi_u^{-1},\Pi_v^{-1}).$$

For each u, the index stores Π_u^{-1} instead of $\hat{u}$, thus it can compute sim in $O(K)$ time. Thus the total storage space is $Kn\log\rho$ bits[3] (recall that $K=\rho$ in this index, yet we give the complexities for the more general case of keeping the K closest among ρ references). The fact that $\rho=K$ severely limits the number of references used, due to space considerations. For finding the γ candidates, they simply scan the n signatures, comparing them with that of the query. Thus the extra CPU time is $O(Kn)$, which makes scalability an issue even after some speedups achieved in subsequent work [21,52]. Building the index requires ρn distance computations, plus $O(n\rho\log K)$ time to find the K smallest distances to each element. The inverse permutations are built in $O(Kn)$ additional time.

Overall, the PI achieves excellent quality of results with few distance computations, but it is useful only with relatively small databases where the distance function is costly to compute.

3.1.2. Brief Permutation Index (BPI)

Introduced by Tellez et al. [53], this index encodes the permutations of the PI in a much more compact form, which however loses some information. Each signature is encoded as a bitvector of length K. The encoding is based on measuring the displacement of the permutants with respect to the identity. That is, when $\Pi_u^{-1}(i)$ differs from i by more than m (for a parameter m) the ith bit in the signature is set to 1, else to 0. The dissimilarity between two signatures is their Hamming distance.

As a result, the index is much smaller and faster than the PI: it requires only Kn bits, and the candidates can be obtained in $O(Kn/w)$ time on a RAM machine of w bits. The downside is that the similarity is coarser, and thus a larger γ, and even a larger K, might be needed to obtain the same result quality. The construction cost is the same as for the PI.

They achieve good results with $m=\rho/2$. Further, they observe that the central permutants do not usually displace much, thus the central band of the signatures contains mostly 0's. Hence they remove the $\rho/2$ bits in the middle of the bitmaps, halving the space.

Although computing the Hamming distance is way faster than with the PI, a sequential scan can still be too costly. They presented later a variant where the signatures themselves are indexed with Locality Sensitive Hashing [52]. The speed increases noticeably, but since this indexed search for candidates is now done approximately, the quality of the result is degraded (also, there is no guarantee on such quality).

The BPI also fits in the $\mathsf{K}-\mathsf{nr}$ framework. Conceptually, the signatures are still permutations, but the similarity function between two permutations only considers the following:

$$\mathsf{sim}_{\mathsf{BPI}}(\hat{u},\hat{v})=\sum_{i=1}^{K}\begin{cases}0 & \text{if } [|\Pi_u^{-1}(i)-i|\le m]\ne \\ & [|\Pi_v^{-1}(i)-i|\le m],\\ 1 & \text{otherwise},\end{cases}$$

where $[c]$ is 1 if logical condition c holds and 0 if not. Now, given that sim only needs limited information about the permutation, the index can be stored within much less space, namely we only need to store, for each $\Pi_u(i)$, the bit $[|\Pi_u^{-1}(i)-i|\le m]$. Then $\mathsf{sim}_{\mathsf{BPI}}$ is K minus the Hamming distance between the two bitmaps.

3.1.3. Metric Inverted File (MIF)

Amato and Savino [2,1] presented a scalable version of the PI. They detach the roles of K and ρ, so that the signature $\hat{u}$ stores only the ordering of the K permutants closest to u. This allows using a much larger value for ρwhile the memory usage stays proportional to Kn. The similarity sim is computed as a Spearman Footrule where some information is missing, namely, we only know the differences in the positions of the permutants that are present in both signatures. Otherwise, they use a penalty of ω, which can be set to, say, ρ:

$$\mathsf{sim}_{\mathsf{MIF}}(\hat{u},\hat{v})=\omega K-\sum_{i=1}^{K}\begin{cases}|i-j| & \text{if } \hat{u}(i)=\hat{v}(j)\\ \omega & \text{if } \hat{u}(i)\notin\hat{v}.\end{cases}$$

If the signatures are stored as such, the space is $Kn\log\rho$ bits, but the computation of sim takes $O(K\log K)$ time. If, instead, $\hat{u}$ is stored with the references in increasing identifier order, sim can be computed in $O(K)$ time. In exchange, we need to store the reordered references and their original positions in $\hat{u}$, which requires $Kn\log(K\rho)$ bits.

However, the authors go much further. They show that an inverted index [6] can be used to avoid scanning the signatures that have no elements in common with $\hat{q}$. More precisely, for each reference $r\in R$, they store a list of the elements $u\in S$ where $r\in\hat{u}$. For each such u, the list stores the pair (u,i) such that $r=\hat{u}(i)$. The signatures themselves need not be stored. The total space is dominated by the $Kn\log(Kn)$ bits used by the lists.

At query time we only need to scan the lists of the references $r\in\hat{q}$. A data structure for set membership is initialized. For each pair (u,i) found in the list of reference $r=\hat{q}(j)$ we search for u in the structure. If it is not found, we insert it with score $\omega-|i-j|$; otherwise we add $\omega-|i-j|$ to its current score. At the end, we collect the γ elements u with highest score (equal to $\mathsf{sim}(\hat{u},\hat{q})$), and this forms the candidate set. If the data structure is a binary tree, the CPU time is $O(N\log n)$, where $N\le Kn$ is the number of times the elements of $\hat{q}$ appear in the signatures of the database. A hash table, instead, obtains $O(N)$ average time.

[3] Our logarithms are in base 2 by default.

3.1.4. Prefix Permutations Index (PPI)

This approach [18,19] is a variant of MIF that also stores the K-length prefix of the permutation, but uses a different sim function. The similarity $\mathsf{sim}(\hat{u},\hat{v})$ is simply the length of the prefix shared by $\hat{u}$ and $\hat{v}$:

$$\mathsf{sim}_{\mathsf{PP}}(\hat{u},\hat{v}) = \max\{i \in [0,K], \hat{u}[1..i] = \hat{v}[1..i]\}.$$

The space usage can be as low as $Kn \log \rho$ bits and the similarity function can be computed in $O(K)$ time (and usually less). In exchange, the similarity function is very coarse, which leads to low recall in the answers.

The authors show that much faster CPU times are obtained if the prefixes are stored in a compressed trie data structure [6]. In this case a query q can be solved by traversing the trie following the symbols of $\hat{q}$ and exploring the nodes found in decreasing depth order (i.e., from longest to shortest shared prefixes). The compact trie is usually smaller than MIF's inverted index.

3.1.5. Compressed Neighborhood Approximation Inverted Index (CNAPP)

All approaches described above use the order of the K nearest neighbors to estimate proximity. Tellez et al. [54] disregard the order and only consider the set itself. It is convenient to define

$$\mathsf{set}(\hat{u}) = \{r \in [1..\rho], \exists i \in [1..K], \hat{u}(i) = r\}.$$

The similarity between two objects is defined as the cardinality of the intersection of the K nearest neighbors of the objects:

$$\mathsf{sim}_{\mathsf{CNAPP}}(\hat{u},\hat{v}) = |\mathsf{set}(\hat{u}) \cap \mathsf{set}(\hat{v})|.$$

By storing the signature in increasing order of reference identifier, the intersection can be computed in $O(K)$ time and the space to store the signature can be reduced to $nK \log(\rho/K) + O(nK)$ bits. It is also possible to use an inverted index exactly as in MIF, with the advantage that the lists need only store the element u that contains each reference r and not the position where r appears in $\hat{u}$. Therefore the index requires $nK \log n$ bits. Various techniques to compress the index, for example reordering the objects, are explored.

3.2. New proximity indexes from the $\mathsf{K-nr}$ framework

From the discussion above, it is apparent that there are many more alternatives to explore. All it takes is to define a similarity function between sets of K elements and we can have a new proximity index with improved time performance and/or answer quality.

Observe that the reviewed indexes can be regarded as taking the $\mathsf{K-nn}$ signature either as vector (PI, BPI, MIF), or as a string (PP), or as a set (CNAPP). We now explore some alternative signatures and similarity functions on vectors, strings, and sets.

3.2.1. Vectors

For vectors we tried two additional distance functions. The first one is similar to that of MIF, but we use the partial L_2 distance, instead of the partial L_1, over the inverse of the signature permutations. That is, it is equivalent to a Spearman Rho similarity over the partial permutations:

$$\mathsf{sim}_{\mathsf{K-nr}-Rho}(\hat{u},\hat{v}) = \omega^2 K - \sum_{i=1}^{K} \begin{cases} (i-j)^2 & \text{if } \hat{u}(i) = \hat{v}(j) \\ \omega^2 & \text{if } \hat{u}(i) \notin \hat{v}. \end{cases}$$

Our second similarity function is loosely inspired in the cosine similarity of the vector space model of information retrieval [6]. If we regard R as a vocabulary and S as a set of documents, we can define the weight of a reference $r \in R$ for an object $u \in S$ as follows:

$$w_r(\hat{u}) = \begin{cases} (K-i+1)/K & \text{if } r = \hat{u}(i) \\ 0 & \text{otherwise}. \end{cases}$$

Then the closer a reference r to an object u, the closer to 1 will $w_r(\hat{u})$ be. The far references, and those not appearing in $\hat{u}$, will have weight zero or close to zero. The similarity can then be computed as a usual cosine similarity:

$$\mathsf{sim}_{\mathsf{K-nr-Cosine}}(\hat{u},\hat{v}) = \frac{\sum_{r=1}^{\rho} w_r(\hat{u}) w_r(\hat{v})}{((K+1)/2)^2},$$

where we note that weights w_r can be computed from the signatures, and therefore we need to store just the K nearest references to each u. It is interesting to notice that the index can be implemented with an inverted index just as MIF, that is, storing for each $r \in R$ the list of tuples $(u, w_r(\hat{u}))$ for which $w_r(\hat{u}) > 0$. The time complexity is of the same order of the MIF index, but the space usage is $Kn \log(K\rho)$ bits. An inverted index raises the required space to $nK \log(Kn)$ bits.

3.2.2. Strings

The PPI regards the signatures as strings, and uses the longest shared prefix as the measure of similarity. While convenient in terms of CPU time, this similarity measure is very crude and achieves low recall. We introduce the use of more refined similarity functions between strings, which are however costlier to compute.

The *longest common subsequence* (*LCS*) between two strings is the length of the longest string that is a subsequence of both strings. It leads to the following similarity function, which ranges between 0 and K:

$$\mathsf{sim}_{\mathsf{LCS}}(\hat{u},\hat{v}) = LCS(\hat{u},\hat{v}).$$

A slightly more refined function is the *Levenshtein distance (Lev)*, which is the minimum number of symbols that must be inserted, deleted, or substituted in two strings to make them equal. This leads to the following similarity function:

$$\mathsf{sim}_{\mathsf{Lev}}(\hat{u},\hat{v}) = K - Lev(\hat{u},\hat{v}),$$

where we note that K is the maximum value $Lev(\hat{u},\hat{v})$ can take, and thus $\mathsf{sim}_{\mathsf{Lev}}$ ranges between 0 and K. Both measures can be computed in time $O(K^2)$ and are explained in depth in many books and surveys [37]. The space complexity of the index is $nK \log \rho$ in both cases because we store signatures of all objects.

3.2.3. Sets

CNAPP is an index that measures similarity by considering the symbols shared between $\hat{u}$ and $\hat{v}$, therefore regarding them as sets. CNAPP simply uses $\mathsf{sim}_{\mathsf{CNAPP}} = |\mathsf{set}(\hat{u}) \cap \mathsf{set}(\hat{v})|$, which takes values in $[0..K]$. Other known measures of similarity between sets used in information retrieval tasks [27,6] are the Jaccard coefficient,

$$\mathrm{sim}_{\mathrm{Jaccard}}(\hat{u},\hat{v}) = \frac{|\mathrm{set}(\hat{u}) \cap \mathrm{set}(\hat{v})|}{|\mathrm{set}(\hat{u}) \cup \mathrm{set}(\hat{v})|},$$

and the Dice coefficient,

$$\mathrm{sim}_{\mathrm{Dice}}(\hat{u},\hat{v}) = \frac{|\mathrm{set}(\hat{u}) \cap \mathrm{set}(\hat{v})|}{|\mathrm{set}(\hat{u})| + |\mathrm{set}(\hat{v})|},$$

both giving values in $[0,1]$ and being easy to compute in $O(K)$ time if we represent $\mathrm{set}(\hat{u})$ as a sequence with the symbols of $\hat{u}$ in increasing order of reference identifier. Since, however, in our scenario it holds $|\mathrm{set}(\hat{u})| = |\mathrm{set}(\hat{v})| = K$, and $|\mathrm{set}(\hat{u}) \cup \mathrm{set}(\hat{v})| = |\mathrm{set}(\hat{u})| + |\mathrm{set}(\hat{v})| - |\mathrm{set}(\hat{u}) \cap \mathrm{set}(\hat{v})| = 2K - |\mathrm{set}(\hat{u}) \cap \mathrm{set}(\hat{v})|$, both measures are monotonic with the basic CNAPP index.

4. Indexing $\mathrm{K-nr}$ sequences

In this section we introduce a novel representation that is general for the $\mathrm{K-nr}$ framework, and combines space efficiency with efficient filtering of the candidates. Note that, in previous work, one must choose between storing the sequences in raw form, generally using $Kn \log \rho$ bits but requiring $\Omega(Kn)$ time to find the γ candidates, or using an inverted index, which is much faster to spot the potential candidates but increases the space requirement to at least $Kn \log n$ bits.

We propose the use of an *Index on Sequences (IoS)* to obtain both benefits, space and time, simultaneously. Given the n signatures $\{\mathrm{K-nn}(u), u \in S\}$, where we assume each $u \in S$ is identified with a number $i \in [1..n]$, we define a string representation for the index:

$$T = \mathrm{K-nn}(u_1) \cdot \mathrm{K-nn}(u_2) \cdot \ldots \cdot \mathrm{K-nn}(u_n),$$

where $u_i \in S$ denotes the element whose identifier is $i \in [1..n]$ and "$\cdot$" represents the string concatenation. Then T is a string of length $N{=}Kn$ over alphabet R, which we identify with the interval $[1..\rho]$.

A plain representation of string T requires $N\log\rho$ bits. To support inverted index functionality, and other operations useful for the $\mathrm{K-nr}$ framework, we need a representation of T that efficiently answers these queries:

- $\mathrm{Access}(T,i)$ retrieves $T[i]$.
- $\mathrm{Rank}_r(T,i)$ counts how many times $r \in [1..\rho]$ occurs in $T[1..i]$.
- $\mathrm{Select}_r(T,j)$ returns the position of the jth occurrence of $r \in [1..\rho]$ in T.

There are several proposals in the literature [26,25,20,7,8] that offer different space/time tradeoffs for representing an IoS. For this problem, we use a novel IoS representation described in the Appendix, which uses $NH_0(T) + O(N)$ bits of space. Here, $H_0(T)$ is the zero-order empirical entropy of T, that is, $H_0(T) = \sum_{r \in R}(N_r/N) \log N/N_r$, where N_r is the number of occurrences of r in T. In the worst case, when the symbols distribute uniformly, $N_r = N/\rho$ and $nH_0(T) = N \log \rho$. This representation supports Select in constant time and Rank and Access in $O(\log n)$ time.

We describe now how this compressed representation is used to efficiently support the various known and new indexes we have described within the $\mathrm{K-nr}$ framework. It should be clear that indexes based on a sequential traversal of the signatures are easily simulated using Access operations on T, since it holds

$$\hat{u}_i(j) = \mathrm{Access}(T, K \cdot (i-1) + j).$$

However, Rank and Select operations enable the simulation of much more sophisticated traversals.

4.1. $\mathrm{K-nr-MIF}$, $\mathrm{K-nr-Spearman-Rho}$, *and* $\mathrm{K-nr-Cosine}$

An inverted index, like the one used in MIF to speed up queries, is easily simulated using an IoS. Note that the inverted list of a reference $r \in [1..\rho]$ contains the database elements u_i such that r appears in $\hat{u}_i$. Therefore, the jth element of the list of r can be simply computed as

$$List(r)[j] = \lceil \mathrm{Select}_r(T,j)/K \rceil,$$

since $\mathrm{Select}_r(T,j)$ is the position of the jth occurrence of r in a signature, and we take the signature number instead of the position in T. Moreover, the position of reference r in the corresponding signature is

$$\mathrm{Select}_r(T,j) - (List(r)[j]-1)K.$$

The length of the list is $\mathrm{Rank}_r(T,N)$. Algorithm 1 describes the MIF procedure using these primitives; we call the result $\mathrm{K-nr-MIF}$. The procedure for $\mathrm{K-nr-Spearman\ Rho}$ is analogous (just replace line 8 with $v \leftarrow \omega^2 - (j-o)^2$).

Algorithm 1. $\mathrm{K-nr-MIF}$: MIF algorithm with an IoS.

```
Input: The query signature $\hat{q} = \mathrm{k-nn}(q)$, the sequence $T$, the penalty
    constant $\omega$, and the desired number of candidates $\gamma$.
Output: The set of candidate objects.
1:  $A \leftarrow$ hash table storing object identifiers $i \in [1..n]$ as keys and
    partial similarities as values.
2:  for all $j \in [1..K]$ do
3:    $r \leftarrow \hat{q}[j]$
4:    for all $s \in [1..\mathrm{Rank}_r(T,N)]$ do
5:      $p \leftarrow \mathrm{Select}_r(T,s)$
6:      $i \leftarrow \lceil p/K \rceil$
7:      $o \leftarrow p - (i-1)K$
8:      $v \leftarrow \omega - |j-o|$
9:      if $A$ contains key $i$, with value $a$ then
10:       Change its value to $a+v$
11:     else
12:       Insert key $i$ and value $v$ to $A$
13:     end if
14:   end for
15: end for
16: Return the $\gamma$ keys with highest values in $A$
```

To implement $\mathrm{K-nr-Cosine}$, we only need a few changes. The denominator constants can be ignored, as they divide all values by the same number. Now we can use the same algorithm as for $\mathrm{K-nr-MIF}$, with line 8 changed to $v \leftarrow (K-o+1) \cdot (K-j+1)$. At the end, we return the γ keys u with highest values.

4.2. PPI

Algorithm 2. $\mathsf{K}-\mathsf{nr}-\mathsf{PPI}$: PPI algorithm with an IoS.

```
Input: The query signature $\hat{q}=\mathsf{k}-\mathsf{nn}(q)$, the sequence $T$, the
    penalty constant $\omega$, and the desired number of candidates $\gamma$.
Output: The set of candidate objects.
    $C \leftarrow \varnothing$
    $r \leftarrow \hat{q}[1]$
    for all $j \in [1..\mathsf{Rank}_r(T,N)]$ do
        $p \leftarrow \mathsf{Select}_r(T,j)$
        if $p \bmod K = 1$ then
            $C \leftarrow C \cup \{p\}$
        end if
    end for
    for all $i \in [2..K]$ do
        $r \leftarrow \hat{q}[i]$
        $C' \leftarrow \varnothing$
        for all $p \in C$ do
            if $\mathsf{Access}(T,p+i-1)=r$ then
                $C' \leftarrow C' \cup \{p\}$
            end if
        end for
        if $|C'| \le \gamma$ then
            break
        end if
        $C \leftarrow C'$
    end for
    Return $C'$ and $\gamma-|C'|$ further elements from $C/C'$
```

Algorithm 2 describes an implementation of this index using an IoS; we call the result $\mathsf{K}-\mathsf{nr}-\mathsf{PPI}$. We use Select to identify all the signatures that share the first symbol with $\hat{q}$. Then we subsequently filter the set with the next symbols $\hat{q}[i]$, until the resulting set becomes smaller than desired. At this point, all the best γ candidates, sharing a prefix of length i or more with $\hat{q}$, are in C'. If there are less than γsuch candidates in C', the next best ones are still in C/C'.

4.3. $\mathsf{K}-\mathsf{nr}-\mathsf{Jaccard}$, $\mathsf{K}-\mathsf{nr}-\mathsf{LCS}$, *and* $\mathsf{K}-\mathsf{nr}-\mathsf{Lev}$

To compute the cardinality of intersections for CNAPP, we use a variant of the inverted index based procedure used for $\mathsf{K}-\mathsf{nr}-\mathsf{MIF}$, where in this case we simply increment the values associated to the documents u each time we find that $\hat{u}$ contains another reference present in $\hat{q}$. Algorithm 3 gives the pseudocode for this index, which we call $\mathsf{K}-\mathsf{nr}-\mathsf{Jaccard}$.

Algorithm 3. $\mathsf{K}-\mathsf{nr}-\mathsf{Jaccard}$: CNAPP algorithm with an IoS.

```
Input: The query signature $\hat{q}=\mathsf{k}-\mathsf{nn}(q)$, the sequence $T$, the
    penalty constant $\omega$, and the desired number of candidates $\gamma$.
Output: The set of candidate objects.
$A \leftarrow$ hash table storing object identifiers, $i \in [1..n]$ as keys and
    partial similarities as values.
for all $j \in [1..K]$ do
    $r \leftarrow \hat{q}[j]$
    for all $s \in [1..\mathsf{Rank}_r(T,N)]$ do
        $i \leftarrow \lceil \mathsf{Select}_r(T,s)/K \rceil$
        if $A$ contains key $i$, with value $a$ then
            Change its value to $a+1$
        else
            Insert key $i$ and value 1 to $A$
        end if
    end for
end for
Return the $\gamma$ keys with highest values in $A$
```

It is not easy to effectively filter for similarities $\mathsf{K}-\mathsf{nr}-\mathsf{LCS}$ and $\mathsf{K}-\mathsf{nr}-\mathsf{Lev}$ using an IoS. However, we can take advantage of the upper bound

$$LCS(\hat{u},\hat{v}) \le |\mathsf{set}(\hat{u}) \cap \mathsf{set}(\hat{v})|,$$

where the order between symbols is disregarded on the right hand side. Therefore, we can use the same algorithm for $\mathsf{K}-\mathsf{nr}-\mathsf{Jaccard}$, but now sorting the final set of candidates A by decreasing value. We traverse the consecutive candidates u in the sorted A and compute $LCS(\hat{u},\hat{q})$ for each. Those candidates are stored in a final result set C sorted by decreasing value of $LCS(\hat{u},\hat{q})$ and retaining only the largest γ values found. If, at any moment, $|C|=\gamma$ and the least value stored in C is not smaller than the next upper bound to review in A, we can safely stop and return C. Algorithm 4 gives the pseudocode. The filtering procedure for $\mathsf{sim}_{\mathsf{Lev}}$ is exactly the same, as the same upper bound applies.

Even when this scheme is powerful enough to implement cascade filters, in our case the signatures $\hat{u}$ must be reconstructed from T using Access in order to apply LCS. We use a faster algorithm that uses the positions found with Select to partially reconstruct the signatures, replacing those references not in the query by a dummy reference. For this sake, when building the set A we replace component a in the pairs (i,a) by a table $mask$ of $K \times K$ bits, so that $mask(j)[o]=1$ iff reference $\hat{q}[j]$ appears at position o in $\hat{u}_i$. This is precisely the precomputed table needed by a fast bit-parallel LCS computation algorithm [29], which we integrate in Algorithm 5.

It is not hard to see the similarity between Algorithms 3 and 5. We combine both methods into a new variant, called $\mathsf{K}-\mathsf{nr}-\mathsf{JaccLCS}$, that uses

$$\mathsf{sim}_{\mathsf{JaccLCS}}(\hat{u},\hat{v}) = LCS(\hat{u},\hat{v})/K + |\mathsf{set}(\hat{u}) \cap \mathsf{set}(\hat{v})|,$$

which is easily computed by storing both a and $mask$ in the tuples associated to key i.

Algorithm 4. $\mathsf{K}-\mathsf{nr}-\mathsf{LCS}$ algorithm with an IoS.

```
Input: The query signature $\hat{q}=\mathsf{k}-\mathsf{nn}(q)$, the sequence $T$, and the
    desired number of candidates $\gamma$.
Output: The set of candidate objects.
$A \leftarrow$ final $A$ set of Algorithm 3
Sort $A$ by decreasing value
$C \leftarrow \varnothing$ (a min-priority queue of maximum size $\gamma$)
for all (key $i$, value $a$) $\in A$ do
    if $|C| \ge \gamma \wedge min(C) \ge a$ then
        break
    end if
    $l \leftarrow LCS(\hat{u}_i,\hat{q})$
    $C \leftarrow C \cup \{(u_i,l)\}$ with key $l$
    if $|C| > \gamma$ then
        Remove key $min(C)$ from $C$
    end if
end for
Return $C$
```

Algorithm 5. $\mathsf{K}-\mathsf{nr}-\mathsf{LCS}$ algorithm with an IoS. Bitwise operations & (and) and | (or) are used, and a^b means b repetitions of bit a. Sometimes $mask(j)$ is used as a single K-bit value.

```
Input: The query signature $\hat{q}=\mathsf{k}-\mathsf{nn}(q)$, the sequence $T$, and the
    desired number of candidates $\gamma$.
Output: The set of candidate objects.
    $A \leftarrow$ hash table storing partial signatures, $i \in [1..n]$ as keys and a
    small table of $K$ strings of $K$ bits as values
```

```
for all $j \in [1..K]$ do
  $r \leftarrow \hat{q}[j]$
  for all $s \in [1..\mathrm{Rank}_r(T,N)]$ do
    $p \leftarrow \mathrm{Select}_r(T,s)$
    $i \leftarrow \lceil p/K \rceil$
    $o \leftarrow p-(i-1)K$
    if $A$ does not contain key $i$ with table $mask$ then
      Insert key $i$ with a table $mask$ with all zeros
    end if
    $mask(j)[o] \leftarrow 1$
  end for
end for
for all $(i, mask) \in A$ do
  $V \leftarrow 1^K$
  for all $j \in [1..K]$ do
    $t = V \ \& \ mask(j)$
    $V \leftarrow ((V+t)|(V-t))$
  end for
  $llncs \leftarrow 0$ {The length of the longest common subsequence}
  while $V \neq 0$ do
    $V \leftarrow V \ \& \ (V-1)$
    $llncs \leftarrow llncs+1$
  end while
  Associate $llncs$ to key $i$ in $A$
end for
Select the final set of items as in Algorithm 4
```

5. Experimental methodology

The main outcome of our work is a set of indexes for proximity searching. Our experiments will show that they successfully deal with various demanding real-world scenarios. We measure the cost of our indexes in terms of the following parameters:

- *The number of distances computed to answer a query*: This is at most $\rho+\gamma$, but we reduce the ρ component by using an index that allows finding the K nearest references to query q better than with a linear scan (this enables using much larger ρ values). The structure is a variant of the Spatial Approximation Tree (SAT) [38] where the construction algorithm is changed so that the candidates to neighbors are inserted from farthest to closest to the node [15]. Note that this index is built only on the set R of references, not on the whole dataset, so it is small.
- *The CPU time*: Even when this depends on many parameters (e.g., hardware, programming language, compilers, operating system, etc.), it gives a clear real-life sense of the performance for practitioners. Under similar testing conditions, it is an important way to compare indexes without disregarding the cost to find the candidate sets.
- *The memory usage*: The storage requirement is an important factor to consider in practice. Many metric indexes care little about space usage, storing several integers per object. Most of our indexes, instead, use just a few bits per object, allowing the storage of much larger databases in main memory.
- The *quality* of the results, measured in terms of recall.

In all our queries we have used the value $k=30$, that is, we retrieve the 30 nearest neighbors of the queries, as this is a common value in multimedia information retrieval scenarios.

5.1. Developing and running environment

All the algorithms were written in C, with the Mono framework (http://www.mono-project.org). Algorithms and indexes are available as open source software in the *natix* library (and http://github.com/sadit/natix/). Unless another setup is indicated, all experiments were executed on a 16 core Intel Xeon 2.40 GHz workstation with 32 GiB of RAM, running CentOS Linux. The entire databases and indexes were maintained in main memory and without exploiting multiprocessing capabilities of the workstation.

5.2. Description of the datasets

In order to give a rich description of the behavior of our techniques, we select several real-world databases and generate synthetic ones, as detailed below. It is worth noticing that even if our datasets are vector spaces, we never use the coordinates to discard elements; we use the distance as a black box. This allows us to work with the data while disregarding its representation; all we need is a distance function to index the data.

- Documents: This database is a collection of 25,157 short news articles from TREC-3 collection of the Wall Street Journal 1987–1989. We use their tf-idf vectors, taken from the SISAP project, www.sisap.org. We use the angle between the vectors as the distance measure [6]. We remove 100 random documents from the collection and use them as queries (thus these 100 documents are not indexed in the database). The objects are vectors of hundred thousands coordinates. Fig. 2a shows the histogram of distances. This dataset has a very high intrinsic dimension in the sense of Chavez et al. [16], that is, $\mu/2\sigma^2$ where μ is the mean and σ is the standard deviation of the histogram, see Table 1. Even finding the nearest neighbor of a query requires reviewing the entire database in most exact metric indexes. As a reference, a sequential scan needs 0.23 s.
- Colors: This is a set of 112,682 color histograms (112-dimensional vectors) from SISAP, using the L_2 distance. To obtain the query set, we chose 200 histogram vectors at random. Fig. 2b shows the histogram of distances. A harder scenario, Colors-hard, is obtained by retaining the same space but applying a perturbation of ± 0.5 to one random coordinate of each query. Fig. 2c shows the histogram of distances. Note that the difficulty comes from perturbing the query set, since the perturbation is a third of the maximum distance. The intrinsic dimension grows four times compared to queries extracted from the dataset without perturbations (the Colors dataset), see Table 1. A sequential scan on this database takes 0.064 s.
- CoPhIR:This is a 10-million-object subset of the CoPhIR database [11]. Each object is a 208-dimensional vector and we use the L_1 distance. Each vector is a linear combination of five different MPEG7 vectors [11]. We remove 200 vectors at random from the database to use them as queries. Fig. 2d shows the histogram of distances. The complexity of this database comes from its

size, since the concentration around the mean is not so large. A sequential scan takes approximately 23 s.

- RVEC: In order to study the performance as a function of the dimensionality of the datasets, we generate random datasets over six different dimensions: 4, 8, 12, 16, 20, and 24 coordinates. These databases are named with the pattern RVEC-*-1M and use L_2 distance. Each vector was generated randomly in the unitary hypercube of the appropriate dimension. Table 1 shows that their intrinsic dimension is larger than twice the number of explicit coordinates. The query set contains 200 vectors generated at random, so query sets and datasets are disjoint with high probability.

6. Empirical tuning and performance results of K − nr

6.1. Space usage

Comparing two indexes with the same number of references is not fair when their space usage per reference is very different. Fig. 3 is a log–log plot that illustrates how different are the scales of memory usage across different indexes on CoPhIR. The results are similar with the other collections.

First, it is clear that PI can only be used on very small databases. It uses over 300 times more memory than the most compact indexes when the number of references grows. This is because it stores ρn values, as opposed to just Kn (we use K=7, which will turn out to be a good value). With a smaller constant, BPI has the same problem. While it takes roughly 10 times less space than PI, this is still about 30 times more space than the most compact indexes.

Fig. 3 also compares the space used by an explicit inverted index (as in the original MIF structure) as opposed to our space-efficient version based on an IoS. On MIF, an explicit inverted index uses 1.3–2.5 more space than an IoS (the inverted index always uses log(Kn) bits per entry, whereas the IoS uses log($K\rho$), thus the space grows slowly towards the right of the plot). The least space demanding indexes are those implemented using an IoS (K − nr−MIF, K − nr−PPI, K − nr−Cosine, K − nr−Jacc, K− nr−LCS, and K − nr−JaccLCS). They do not even need to store the position of the references within the signatures, and thus use only log ρ bits per entry.

6.2. Quality of the results

Fig. 4 shows recall versus search time as the number of references ρ grows from 64 to 2048, with $\gamma = 1\%$ (left) and 3% (right), and K=7. We show one real-life dataset per row. In CoPhIR-10M we use smaller ρ values for indexes with high memory requirements (PI and BPI) in order to keep them in RAM. We show all the described techniques except K − nr−S−Rho, which performed very similarly to K − nr−MIF (called simply MIF in the plots), and K− nr−Lev, which was always inferior to K − nr−LCS. As a control value, we also show the performance of a sequential scan (Seq).

Most of the indexes improve their recall as ρ grows, until reaching around 0.90–0.95. The exceptions are K − nr−PPI (called simply PPI in the plots), which is unable to improve beyond a low recall value, and PI and BPI, which on most datasets do not reach high recall values for lack of available memory. The time, on the other hand, grows with ρ for the indexes that use $K = \rho$ (i.e., PI and BPI). For the other indexes, which use K/ρ, there is a ρ value that yields an optimum time performance.

Indexes PI and BPI are also generally outperformed by most indexes, unless the number of references is very low and so is the recall, an irrelevant niche. An important exception occurs on the CoPhIR dataset, where PI achieves perfect or almost perfect recall, and BPI matches it with $\gamma = 3\%$. Both achieve much better recall (albeit being slower) than the other indexes.

The other indexes require about the same time for the same value of ρ, but K − nr−Cosine (cos) obtains clearly better recall for the same time cost (the difference is more notorious for $\gamma = 1\%$). An exception, on Colors-hard, are K − nr−JaccLCS (jacc-lcs) and K − nr−MIF, which are second-best in the other datasets and here outperform K − nr−Cosine. Seen in terms of time performance, on the other hand, CNAPP generally outperforms the others in time for a given recall, being slower in some cases than K − nr−Jaccard (jacc). K − nr−PPI is even faster for the recall values it reaches, which are unfortunately very low. As γ increases (3%), most methods tend to perform similarly.

The fact that the time performance has an optimum value can be explained as follows. First, as ρ increases, the amount of shared references with $\hat{q}$ decreases, and thus the inverted lists are shorter and less CPU time is needed to find the best γ candidates. On the other hand, for sufficiently large ρ, the number of distance computations, even with the speedup of the SAT structure, becomes dominant and the search becomes slower. This shows up on Documents, where the distance computation is most expensive.

The effect on recall is bit more complicated. Initially, increasing ρ improves recall because it allows the K closest references to different elements to become distinct. However, when ρ grows beyond some limit, the K closest references to different elements start having empty intersection and are not useful to hint closeness. As we can see, however, our structures surpass 0.90 and even 0.95 of recall, within an order of magnitude less time than a sequential scan.

6.3. Performance on increasing intrinsic dimension

Fig. 5 shows the performance when dimension grows. For this experiment we use RVEC-*-1M datasets; statistics and shapes are shown in Table 1 and Fig. 2. In this experiment PPI was left out to improve clarity, focusing on the indexes that produce high recall values. Each subfigure represents a different γ (0.3%, 1%, and 3%). The recall degrades as dimension grows, going from 1.0 in dimension 4 to 0.4–0.8 in dimension 24 (the degradation is slower for larger γ). This is the same phenomenon observed in other approximate indexes: they perform much better than the exact indexes (with low error ratio) on high dimensions, but they also worsen as the

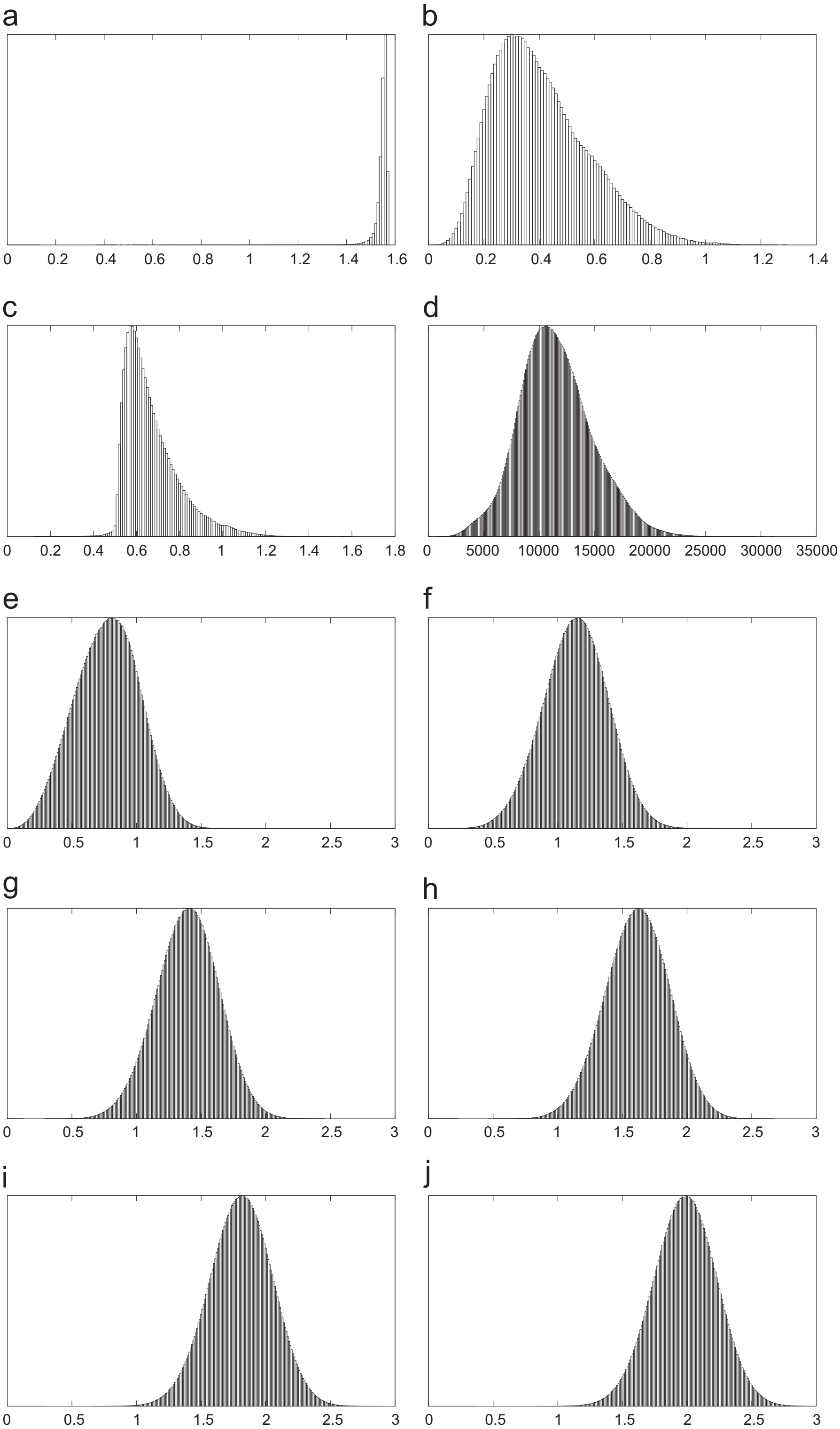

Fig. 2. Histograms of distances of our datasets. (a) Documents. (b) Colors. (c) Colors-hard. (d) CoPhIR. (e) RVEC-4-1M. (f) RVEC-8-1M. (g) RVEC-12-1M. (h) RVEC-16-1M. (i) RVEC-20-1M. (j) RVEC-24-1M.

dimension grows. In the worst scenario, where $\gamma = 0.3$ and dimension 24, K – nr–Cosine, K – nr–MIF and K – nr–JaccLCS perform best, but still obtain low quality. As γ increases to 3%, however, recall improves up to 0.8. However, as γ increases, the search time gets close to a sequential scan.

Table 1
Statistics of our datasets. The mean and the standard deviation, μ and σrespectively, are given relative to the maximum distance, d_{max}. Value $\mu/2\sigma^2$ is the intrinsic dimension as defined by Chavez et al. [16].

Name	$\frac{\mu}{2\sigma^2}$	d_{max}	μ	σ
Documents	982.99	1.57	0.985	0.022
Colors	8.59	1.38	0.302	0.032
Colors-hard	36.32	1.73	0.390	0.073
CoPhIR	19.31	32,682	0.357	0.096
RVEC-4-1M	11.22	1.80	0.426	0.138
RVEC-8-1M	20.21	2.19	0.509	0.112
RVEC-12-1M	29.78	2.55	0.546	0.096
RVEC-16-1M	38.21	2.81	0.580	0.087
RVEC-20-1M	45.39	2.95	0.607	0.082
RVEC-24-1M	55.17	3.21	0.615	0.075

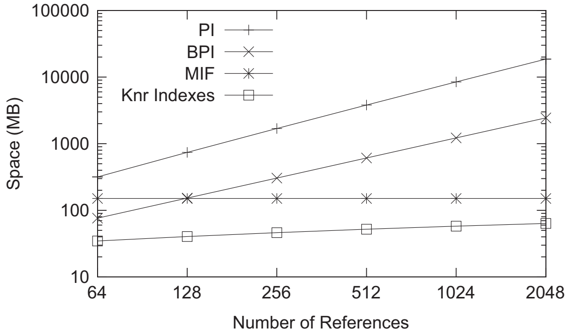

Fig. 3. Memory usage of the indexes on CoPhIR.

6.4. *Performance on increasing dataset sizes*

Fig. 6 compares the performance of the techniques when n increases, on CoPhIR-10M. Each curve plots the performance for $n = 3^5, 10^6, 3 \times 10^6$, and 10^7, dividing the time by that of the sequential scan. An interesting effect as n grows and fixing γ as a percentage of n (1% and 3%) is that recall improves as n grows. The time, instead, stays more or less stable around a small fraction of the sequential search time. As in other experiments, PI and BPI are too slow compared to a sequential scan. The other indexes are much faster and all perform similarly.

6.5. *Varying the length of* $\mathsf{K-nn}(q)$

We have described K as a parameter used at construction and search time. In principle, it is possible to use a signature for query q whose length, $\kappa = |\hat{q}| = |\mathsf{K-nn}(q)|$, is different from K. This can increase CPU time, but not memory (indeed we do not need to rebuild the index to change κ), and it yields further flexibility. While not all similarity functions for signatures are easily adapted to the case $\kappa \neq K$, we study this idea for the particular case of $\mathsf{K-nr}$–Jaccard, where the size of the intersection between signatures of different sizes makes perfect sense, and values K and κ are easily modified at construction and search time, respectively.

Fig. 7 shows the results. The experiment considers fixed K=7 and $\rho = 2048$, and varying $\kappa = 6, 9, 12, 15$, and 18. Each curve in the figure shows the performance for a different γ value (0.1%, 0.3%, 0.45%, 0.6%, 1%, and 3%). We show three real-world datasets and RVEC-16-1M.

It is clear that search time increases with κ, as this makes the construction of $\hat{q}$ slower and also Algorithm 3 (modified to support Jaccard with independent K and κ) performs more iterations. Recall, on the other hand, increases up to an optimum and then decreases again, which can be seen especially for low γ values. This is because many items not really close to q appear in $\hat{q}$, and their intersections with the signatures of the elements introduce noise. There is no clear rule to select κ, but experimental results show that small values are sufficiently good for all the tested datasets. Yet, larger datasets like CoPhIR-10M and RVEC-16-1M make good use of larger κ values. In general, varying κ we obtain combinations of recall and search time that are not possible by just varying γ.

6.6. *Tuning the* $\mathsf{K-nr}$ *indexes*

The flexibility of $\mathsf{K-nr}$ indexes is one of the sources of their great potential, but also may be problematic for an end-user. In this section we provide a simple guide to find a competitive tuning, even if not the optimal one.

Once we have chosen an index from the $\mathsf{K-nr}$ family, we must decide on the values K, ρ, γ, and, if the index permits it, κ.

Parameter γ is the easiest to set. It is related to the amount of work, in terms of distance computations, we are willing to perform. This is usually set to a small percentage of the database, say 1% or 3% as in this paper. Increasing γ always improves the quality of the answer.

Let us now consider parameter K. Since K is the signature length of the objects stored in the database, this parameter impacts directly on the space used by the index (which is $O(Kn)$ words), and also in its CPU speed, since comparing two signatures takes at least $O(K)$ time. If we compare all the signatures directly with that of the query, this implies $O(Kn)$ extra CPU time. If, instead, we index the signatures to avoid the sequential scan, K determines the dimensionality of the mapped space, which also impacts on the CPU time needed to find the suitable candidates.

On the positive side, a larger K improves the recall of the algorithm, up to a certain point. After that point, however, recall does not improve and may even degrade due to noise. In general, due to memory restrictions, K has to be chosen to be a small integer, and the turning point in the recall is not reached. For example, along this paper we have used a fixed K=7, which roughly means storing 7 short integers per database element. This yields a reasonable sized index and a good enough recall.

Parameter ρ impacts on the number of distance computations at query time, as we compute ρ distances to map query q to the space of the references. Since the total number of comparisons is $\rho + \gamma$, it is advisable to set ρ to a small fraction of γ (say, 10–20% of γ), so that the cost to map q is not too significant compared to the γ distance computations invested in improving the answer. Since γ is usually large in absolute terms, this rule of thumb allows for fairly large values of ρ, sufficient to have a representative set of references from where to choose the best K. For

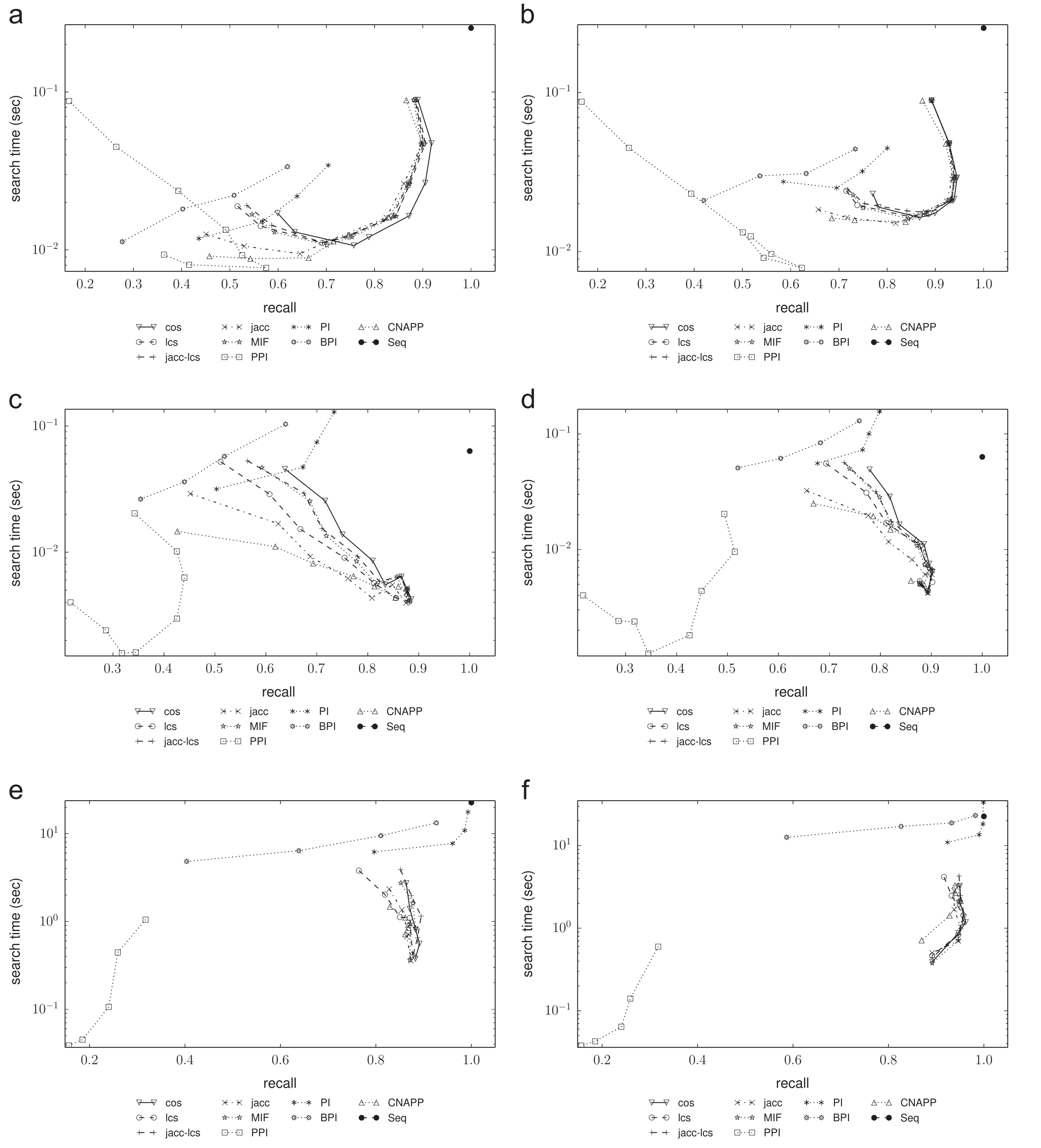

Fig. 4. Search time versus recall as a function of $\rho = 64$, 128, 256, 512, 1024, 2048, 4096, 8192, 16,384, on three real-world datasets. (a and b) Documents with $\gamma = 1\%$ (left) and $\gamma = 3\%$ (right). (c and d) Colors-hard with $\gamma = 1\%$ (left) and $\gamma = 3\%$ (right). (e and f) CoPhIR-10M with $\gamma = 1\%$ (left) and $\gamma = 3\%$ (right). Here PI uses $\rho = 16$, 32, 64, 128, and BPI $\rho = 64$, 128, 256, 512.

example, if we have 1 million objects ($n = 10^6$) and set γ to 3% of n, $\gamma = 30,000$, we can use $\rho = 2000$ references without impacting much (15%) on the total number of distance computations. Along this paper we have many times used $\rho = 2048$.

Finally, we have parameter κ, on indexes that permit using it. This parameter allows us to trade recall for CPU time, for a fixed K value, by using a signature of length $\kappa \neq K$ for the query. Then the CPU time usually becomes at least $O(\kappa n)$, but in exchange we can improve recall without increasing the memory usage (K) nor the number of distance computations ($\rho + \gamma$).

To better choose κ we need a training set of queries Q. The use of such a set to tune indexes is common in the literature, and is typical, for example, in LSH indexes. Given Q, and having chosen the other parameters, we increase κ until we reach the desired recall level on Q, or we reach the maximum allowed extra CPU time. For example, we have experimented with κ values from 6 to 18 in the previous section.

7. Comparison with hashing-based indexes

In this section we compare a good representative of K − nr indexes, K − nr Cosine, with the most prominent state-of-the-art alternatives: indexes based on hashing.

In order to test LSH we used the E^{2}LSH tool,[4] from Alex Andoni. It automatically optimizes an LSH index for the available memory. In order to be fair, since LSH offers distance-based rather than recall-based guarantees, we used it to search by radius (i.e., return any element within distance r to the query), setting r as the average radius of the 30-th nearest neighbor. To avoid extreme or uninteresting cases, we removed queries yielding empty answers and those with more than 1000 answers.

DSH, instead, is a nearest neighbor index, which must be optimized to search for a particular number k of near neighbors. We created the DSH index for the case $k=30$, which is the case we test. We use the implementation distributed by the DSH authors[5] [23].

We manually optimized DSH indexes in two parameters, (i) the hash family width h, and (ii) the number of hashing families L. The goal was to obtain a balance between recall and time. As for other LSH based indexes, the general optimization procedure consists in optimizing h for speed within the available memory and probabilistic constraints. Instead, DSH was not much sensitive to h for our query sets. Thus, for DSH we set h to make the index fast enough, and then increased L to improve the recall rate.

Our comparison testbeds are Colors-hard and CoPhIR-1M, two databases created from real-world processes. We excluded Documents from the comparison because the LSH and DSH implementations only support Euclidean distance. Moreover, according to its authors, LSH [24] makes no sense on datasets following a uniform distribution, thus we also excluded RVEC. We normalize our CoPhIR-1M dataset to have coordinate values between $[-1,1]$ and fix the distance to be L_2 (CoPhIR-1M uses L_1 in our other benchmarks). This involves changing the 16-bit integers that store distances to floating point numbers.

We selected K − nr Cosine to represent K − nr methods; it was constructed with $K=7$ and $\rho=2048$, and κ was set to K in the search step. Construction times are high, but amenable to parallelization. We demonstrate this by parallelizing the construction among the 16 available cores. Queries, instead, always use a single core.

While our implementations are written in C and run under the Mono virtual machine, E^{2}LSH and DSH are written in C++, and run in native form, thus they have an advantage of around 3X. In order to give a full picture of the performance, we report both the absolute times and speedup. The latter is computed with respect to the sequential scan in its own implementation.

As a side note, modifying our datasets from 16-bit integers to floating point numbers improved all the search times, due to internal processor optimizations.

7.0.1. Colors-hard

Table 2 compares the K − nr Cosine with LSH and DSH on the Colors-hard dataset. We use our standard setup for K − nr, which is not optimal (it achieves 0.90 recall). We allow K − nr−cosine to review at most 3% of the database. We use the achieved recall to adjust the probabilities of LSH, that is, we ask to E^{2}LSH to optimize for a success probability of 0.9, a query radius of 0.491 (the average radius of the 30-th NN), and use our Colors-hard queries (removing those not matching our fairness criteria for LSH). We allow E^{2}LSH+ to use 100, 300, and 1000 MB of memory (E^{2}LSH+ was unable to use less memory). For DSH, we set $h=15$, since it was not sensitive to larger h values, thus we were unable to get better search times. We allow DSH to use up to 4 hashing function families, since that matches the performance of K − nr Cosine.

Even with the C-vs-C++ handicap, K − nr Cosine achieves the best absolute search time, being more than 4 times faster than the second-best (LSH-300MB) while using 0.5% of its memory. The third-best, coming close, is DSH $L=1$, which uses slightly less memory than K − nr Cosine, but its extremely low recall renders it useless. K − nr Cosine is also unbeatable in speedup, being almost 13 times faster than a sequential search programmed in C. In fact, no DSH or LSH variant outperforms a sequential search programmed in C++. DSH $L=4$ is able to reach higher recall than K − nr Cosine, yet at the price of reviewing a large fraction of the dataset (over 20%), which translates into high search times (more than 16 times slower than K − nr Cosine).

The preprocessing time needed for K − nr Cosine is about 4 min, which becomes less than 14 s when parallelized over our 16-core hardware.

7.0.2. CoPhIR-1M

Table 3 compares the performance of K − nr Cosine, LSH, and DSH on the CoPhIR-1M dataset. As for Colors-hard, we use a simple parameter selection for K − nr, that is, $K=7$ and $\rho=2048$, based on the experimental study of the previous section. We obtained a recall of 0.954 reviewing 3% of the database. Thus, we fix the success probability of E^{2}LSH to be 0.94, based on average recall of other K − nr methods for $n=10^6$, as shown in Fig. 6. For LSH we used a radius of 0.6112, the average radius of the 30-th NN. We allow E^{2}LSH to use 1000, 3000, and 9000 MB of memory (E^{2}LSH+ was unable to effectively use less or more). For DSH, we set $h=15$ since, as for Colors-hard, DSH does not improve when h increases. We use $L=1$ for DSH, since it already achieves perfect recall, yet at the cost of reviewing 13% of the database, which translates into the highest absolute search time.

In this benchmark, LSH-9000MB achieves the best absolute search time, followed by K − nr Cosine. Precisely, LSH-9000MB is 2.3 times faster than K − nr Cosine, yet it uses 600 times more memory. Note, however, that the

[4] http://www.mit.edu/andoni/LSH/

[5] http://www.comp.nus.edu.sg/dsh/download.html

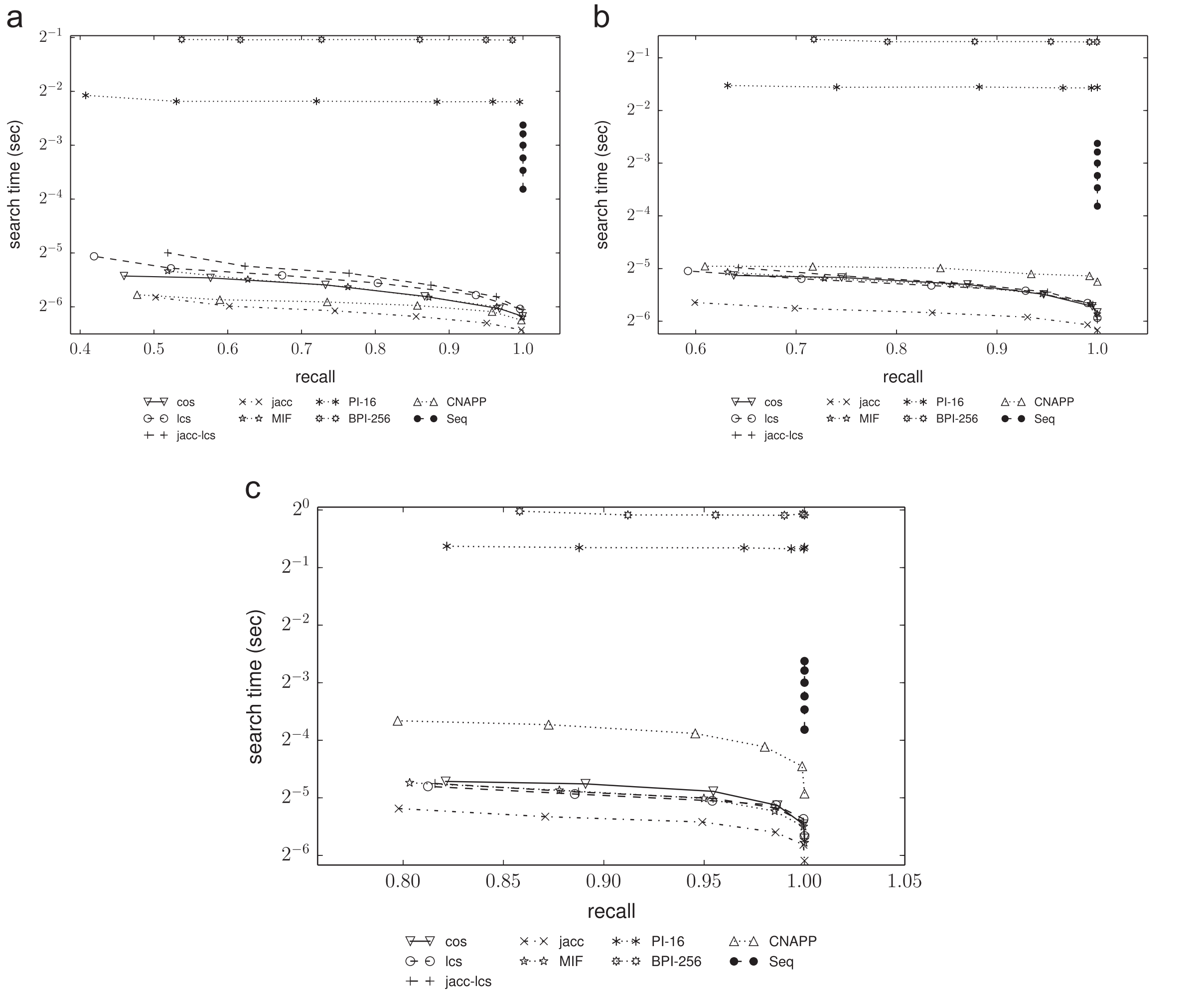

Fig. 5. Performance on increasing dimensionality on random vectors (RVEC-*-1M). Each point at the curves refers to a different dimension value (4, 8, 12, 16, 20, and 24). As the dimension grows, time increases and recall decreases in the curves. We use $K=7$ and $\rho=2048$ (except PI and BPI, which use 16 and 256 references, respectively). (a) $\gamma=0.3\%$. (b) $\gamma=1\%$. (c) $\gamma=3\%$.

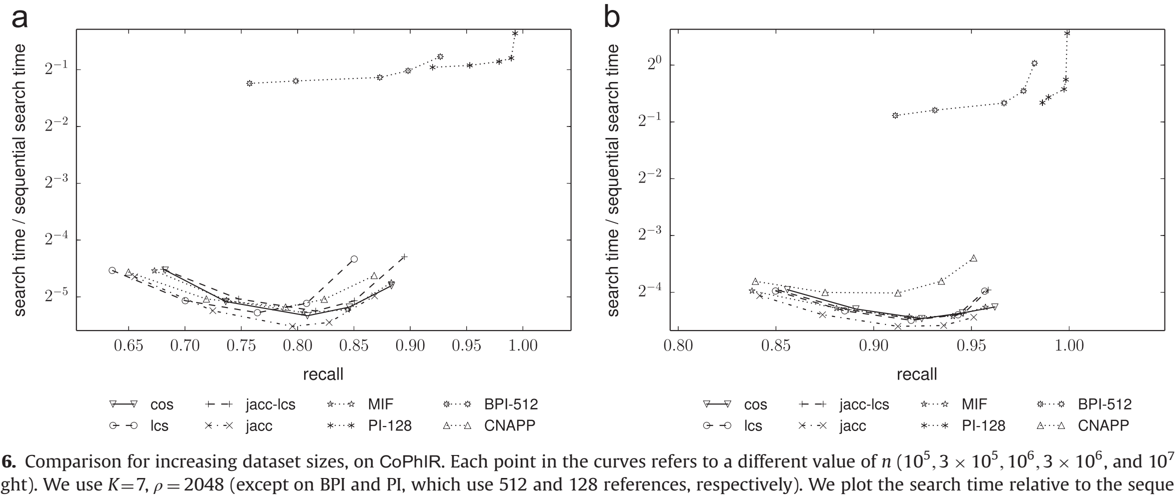

Fig. 6. Comparison for increasing dataset sizes, on CoPhIR. Each point in the curves refers to a different value of n ($10^5, 3\times10^5, 10^6, 3\times10^6$, and 10^7, left to right). We use $K=7$, $\rho=2048$ (except on BPI and PI, which use 512 and 128 references, respectively). We plot the search time relative to the sequential scanning time. (a) $\gamma=1\%$. (b) $\gamma=3\%$.

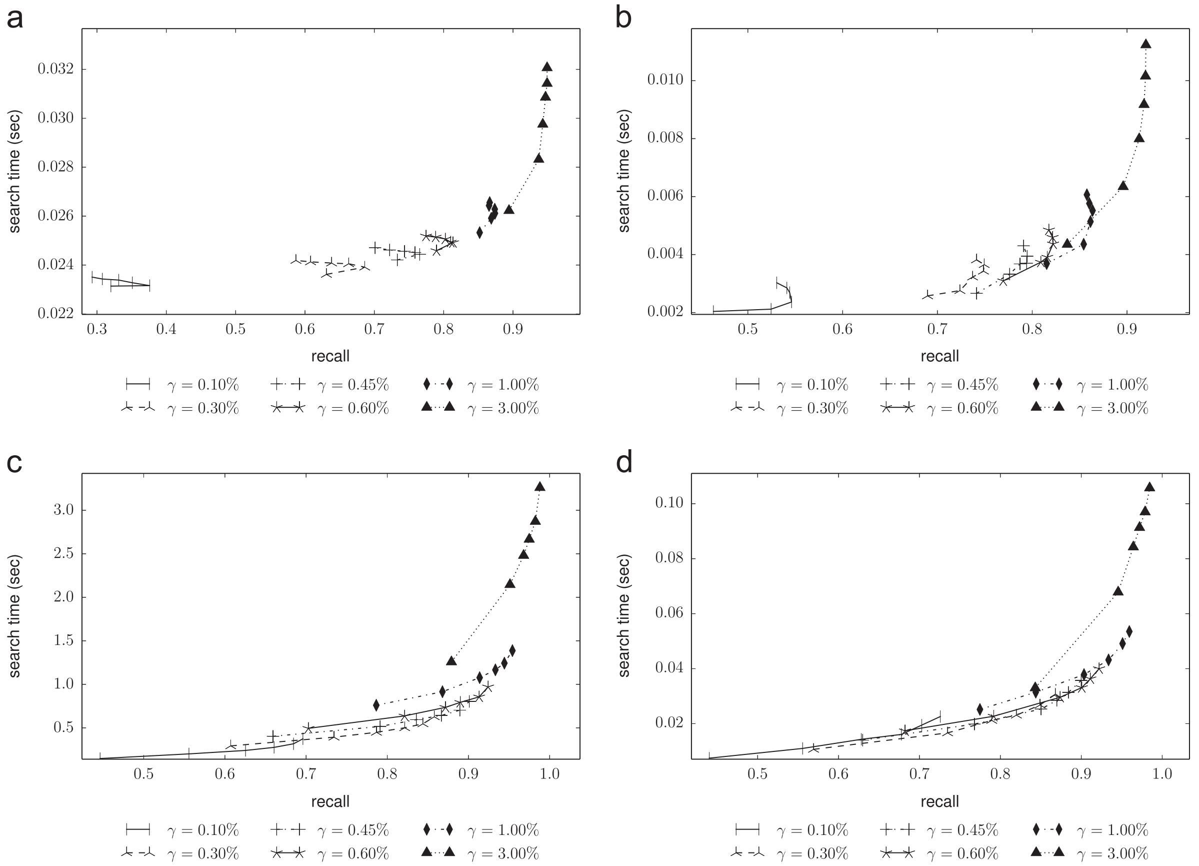

Fig. 7. Performance varying κ and γ on real-world and synthetic datasets. The indexes use $K=7$ and $\rho=2048$. We use $\kappa=6,9,12,15,18$. Increasing κ yields increased time. Each curve corresponds to a a different γ value. (a) Documents. (b) Colors-hard. (c) CoPhIR-10M. (d) RVEC-16-1M.

difference in time performance may be due to the use of C versus C++: in terms of relative times, K − nr Cosine reaches a speedup over 13, which becomes only 9 for LSH-9000.

Finally, the preprocessing time of K − nr Cosine is its worst feature, since it needed 80 min to index the CoPhIR-1M dataset. When parallelized over 16 cores, however, this became 5 min.

8. Summary and perspectives

We have presented a novel framework for approximate proximity search algorithms, called *K* Nearest References (K − nr). The framework consists in choosing a subset *R* of ρ reference objects from the database, and mapping the original proximity problem to a simpler space where each object is represented by a signature built from the *K* nearest references to it. Then a fixed number γ of candidates are obtained according to the similarity between the signatures of the objects and that of the query *q*, and the actual query is directly compared to the obtained candidates.

Our framework encompasses disparate proximity indexes such as the Permutation Index (PI) [14], the Brief Permutation Index (BPI) [53], the Metric Inverted File (MIF) [2], the PP-Index (PPI) [18], and the Compressed Neighborhood Approximation Inverted Index (CNAPP) [54]. We also give several examples of how novel indexes can be easily derived from the framework, by designing new similarity functions from the signatures.

Our second contribution is a generic support for implementing K − nr indexes. We show that by representing the sequence of signatures using a compressed format that answers certain basic queries, we can provide not only direct access to the signatures but also inverted-index functionality, which is useful to speed up a number of K − nr indexes while reducing, instead of increasing, the space usage. This allows recasting various existing indexes that are known to be very effective (e.g., MIF) into a space-efficient representation that allows handling larger databases in main memory. We also study other generic mechanisms to improve K − nr indexes, such as varying the signature size of the query.

Some of the new indexes we design, for example those based on cosine similarity or on a combination of Jaccard coefficient and longest common subsequences between the signatures, turn out to outperform the state of the art in several cases. In particular, we show that hashing based indexes (LSH and DSH), the most clear competitors in this scenario, are outperformed as well: those that slightly outperform our indexes in recall or time, are outperformed by a large margin in either space, time, or recall, depending on the case. The K − nr indexes are slower than LSH and DSH at construction; we have shown that this can be

Table 2
Performance comparison between K − nr Cosine, LSH, and DSH for the Colors-hard dataset. For hashing methods, h is the number of hashing functions, and L the number of families of hashing functions. Column "review" refers to the fraction of the database compared with the query; this is not reported by the LSH software.

Index	Memory (MB)	Preprocessing time	Search time (s)	Speedup	Review	Recall	Parameters	
K − nr cos.	1.7	14 s	0.005	12.793	0.033	0.896	$K=7$	$\rho=2048$
DSH	1.4	35 s	0.022	0.886	0.057	0.295	$L=1$	$h=15$
DSH	2.6	40 s	0.048	0.408	0.120	0.658	$L=2$	$h=15$
DSH	3.8	44 s	0.064	0.310	0.161	0.796	$L=3$	$h=15$
DSH	5.0	50 s	0.084	0.234	0.210	0.931	$L=4$	$h=15$
LSH	100	1 min	0.029	0.691	–	0.886	$L=36$	$h=8$
LSH	300	3 min	0.021	0.926	–	0.700	$L=153$	$h=14$
LSH	1000	4 min	0.024	0.831	–	0.852	$L=595$	$h=20$

Sequential K − nr construction takes 4 min.
The C# seq time is 0.063 s.
The C++ seq time is 0.020 s.
DSH indexes selected its L hashing families among 200 families.
The LSH search radius is 0.491 (average radius of the 30-th NN).
LSH indexes were constructed with a success probability of 0.9.
LSH indexes used 188 queries.

Table 3
Performance comparison between K − nr Cosine, LSH, and DSH for the CoPhIR-1M dataset. For hashing methods, h is the number of hashing functions, and L the number of families of hashing functions. Column "review" refers to the fraction of the database compared with the query; this is not reported by the LSH software.

Index	Memory (MB)	Preprocessing time	Search time (s)	Speedup	Review	Recall	Parameters	
K − nr cos.	15	5 min	0.072	13.694	0.030	0.954	$K=7$	$\rho=2048$
DSH	5.1	10 min	0.849	0.325	0.130	1.000	$L=1$	$h=15$
LSH	1000	5 min	0.318	0.869	–	0.913	$L=15$	$h=4$
LSH	3000	15 min	0.093	2.972	–	0.897	$L=120$	$h=12$
LSH	9000	30 min	0.031	9.033	–	0.892	$L=496$	$h=18$

Sequential K − nr construction takes 80 min.
The C# seq time is 0.980 s.
The C++ seq time is 0.276 s.
DSH indexes selects its L hashing families among 300.
The LSH search radius is 0.6112 (average radius of the 30-th NN).
LSH indexes were constructed with a success probability of 0.94.
LSH indexes used 216 queries.

alleviated by parallelizing the construction across the cores of the processor.

8.1. Future work

We tried other possible improvements that did not give good results. In particular, using several independent indexes did improve recall on low-recall indexes like PPI, but increased the time to a point where techniques like K − nr−Cosine outperformed it in speed and recall using a single index. We believe, however, that improvements are still possible.

We have shown how the choice of the signature and its similarity function impacts not only on the quality of the approximate answers, but also in the space and extra CPU time required by the index. For example, K − nr−MIF achieves good quality, which is matched by K − nr−Jaccard while using less space. However, K − nr−PPI is much faster than both, but the quality of its answers is much lower. The quest for effective similarity measures that can be operated efficiently and with a compact index, say as effective as K − nr−Jaccard but as fast as K − nr−PPI, is open. Also, other representations for the signatures, different from the one we have chosen, may significantly speed up the search. As an example, full-text indexes [39] can represent the signatures even in more compressed form than the way we have studied in this paper, and allow retrieving very fast all the signatures containing a certain substring or prefix. This makes them an excellent choice for K − nr−PPI (in exchange, accessing individual signatures is slower).

We have studied only static scenarios, where the database does not undergo insertions and deletions of objects. Future research will extend our indexes in order to support dynamic scenarios. In principle, inserting and deleting objects in S without affecting the set of references R is rather simple (one can just mark an object as deleted if it turns out to have been chosen as a reference), and therefore supporting a small degree of updates is not problematic. When massive insertions are carried out, however, the structure of the reference set R may become inadequate, and a complete reorganization may be required. Similarly, just marking removed elements may become inadequate upon massive deletions.

Our techniques use only main memory to store both the index and the database. Compression is used to increase the chance of fitting larger databases in main memory. With sufficiently large databases, however, it will be necessary to resort to secondary memory. In this scenario, a classical inverted index performs better than the compressed representation of the signatures we have advocated, and a different data organization is necessary. Various compressed and disk-friendly inverted index organizations are known [56]; studying their applicability to the proximity search scenario is also future work. Parallel and distributed deployments are also a valid choice that deserves attention.

Appendix A. Indexing sequences

There exist several Indexes of Sequences (IoS) with different space/time tradeoffs [26,25,20,7,8]. These represent a sequence $T[1..N]$ over alphabet $[1..\rho]$ while supporting operations Access, Rank, and Select. Note that an IoS acts as a representation of T, as it can recover any $T[i]$ via operation Access. We introduce a novel representation, XLB [51], that turns out to be convenient for our scenario, characterized by a large alphabet size ρ.

A general technique [25] to handle this problem is to reduce it to representations of binary sequences, where the problem is simpler. Consider, for example, a large binary matrix $M[1..\rho][1..N]$, such that $M[r][i] = 1$ iff $T[i] = r$. This matrix has exactly N 1 s out of ρN cells, exactly one 1 per column. Consider further its row-wise concatenation into a single bitvector $P[1..\rho N]$. Then,

$$\mathsf{Rank}_r(T,i) = \mathsf{Rank}_1(P,(r-1)\rho+i) - \mathsf{Rank}(P,(r-1)\rho)$$
$$\mathsf{Select}_r(T,i) = \mathsf{Select}_1(P, i + \mathsf{Rank}(P,(r-1)\rho)) - (r-1)\rho$$

are easily solved by resorting to binary operations on P (note that it is convenient to precompute and store the ρ values $\mathsf{Rank}(P,(r-1)\rho)$). Two problems have precluded going further in this line in existing solutions: (1) operation Access is not easily supported in time less than $O(\rho)$; (2) P has too many 0 s, which complicates its efficient representation.

To support Access, we consider a permutation Π on $[N]$ that simulates an inverted index on M:

$$\Pi(i) = \mathsf{Select}_1(P,i)\text{od } \rho,$$

that is, $\Pi(i)$ tells the column of M (or position in T) where the ith 1 of M lies. Note that Π needs not be stored; any cell can be computed using the formula above. Further, note that $\Pi^{-1}(i)$ tells the rank of the 1 in P that corresponds to $T[i]$. Munro et al. [36] show how, given access to Π in $O(1)$ time, an extra data structure using N/t bits provides access to Π^{-1} in time $O(t\log n)$. Once we know $\Pi^{-1}(i)$, a further binary search among the precomputed values $\mathsf{Rank}(P,(r-1)\rho)$ yields the row of M where $\Pi^{-1}(i)$ falls in $O(\log \rho)$ additional time. Therefore, we answer $\mathsf{Access}(T,i)$ in $O(\log n)$ time using $O(N)$ further bits (note that $N = Kn > \rho$).

The remaining challenge is to represent P efficiently. We use a representation [41] that requires $NH_0(P)+O(N)$ bits, where $H_0(P) = \log(\rho N/N) + O(1) = \log\rho + O(1)$ is the zero-order entropy of the bitvector. This structure, requiring in total $N\log\rho + O(N)$ bits, answers $\mathsf{Select}_1(P,i)$ in $O(1)$ time and $\mathsf{Rank}_1(P,i)$ in time $O(\log \rho)$.

Therefore, we support operations $\mathsf{Rank}_r(T,i)$ in time $O(\log \rho)$, $\mathsf{Select}_r(T,i)$ in time $O(1)$, and $\mathsf{Access}(T,i)$ in time $O(\log n)$, using $N \log \rho + O(N)$ bits of space. To further reduce the space, we can represent each row of M as a separate bitvector of length N and N_r bits set, where N_r is the number of occurrences of symbol r in T. Then the overall space becomes

$$\sum_{r\in[1..\rho]} N_r \log\frac{N}{N_r} + O(N_r) = NH_0(T) + O(N),$$

that is, the representation becomes a zero-order compressed representation of T. In exchange, operation Rank on the bitvector of row r takes time $O(\log(N/N_r))$, which can be bounded by $O(\log n)$.

References

[1] G. Amato, C. Gennaro, P. Savino, Mi-file: using inverted files for scalable approximate similarity search, Multimed. Tools Appl. 71 (3) (2014) 1333–1362.

[2] G. Amato, P. Savino, Approximate similarity search in metric spaces using inverted files. In: Proceedings of the 3rd International Conference on Scalable Information Systems (InfoScale), 2008, pp. 1–10.

[3] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, Commun. ACM 51 (January) (2008) 117–122.

[4] S. Arya, D. Mount, N. Netanyahu, R. Silverman, Y. Wu, An optimal algorithm for approximate nearest neighbor searching in fixed dimensions, J. ACM 45 (6) (1998) 891–923.

[5] F. Aurenhammer, Voronoi diagrams—a survey of a fundamental geometric data structure, ACM Comput. Surv. 23 (3) (1991) 405.

[6] R.A. Baeza-Yates, B.A. Ribeiro-Neto, Modern Information Retrieval, 2nd edition, Addison-Wesley, USA, 2011.

[7] J. Barbay, F. Claude, T. Gagie, G. Navarro, Y. Nekrich, Efficient fully-compressed sequence representations, Algorithmica 69 (1) (2014) 232–268.

[8] D. Belazzougui, G. Navarro, New lower and upper bounds for representing sequences, in: Proceedings of the 20th Annual European Symposium on Algorithms (ESA), 2012, pp. 181–192.

[9] J. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517.

[10] C. Böhm, S. Berchtold, DA. Keim, Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases, ACM Comput. Surv. 33 (3) (2001) 322–373.

[11] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, F. Rabitti, CoPhIR: a Test Collection for Content-based Image Retrieval, CoRR abs/0905.4627v2, 2009, URL: ⟨http://cophir.isti.cnr.it⟩.

[12] L. Boytsov, Indexing methods for approximate dictionary searching: comparative analysis, ACM J. Exp. Algorithm. 16 (1) (2011).

[13] S. Brin, Near neighbor search in large metric spaces, in: Proceedings of the 21st International Conference on Very Large Data Bases (VLDB), 1995, pp. 574–584.

[14] E. Chavez, K. Figueroa, G. Navarro, Effective proximity retrieval by ordering permutations, IEEE Trans. Pattern Anal. Mach. Intell. 30 (9) (2008) 1647–1658.

[15] E. Chavez, V. Ludueña, N. Reyes, P. Roggero, Faster proximity searching with the Distal SAT, in: Proc. 7th International Conference on Similarity Search and Applications (SISAP), Lecture Notes in Computer Science, vol. 8821, 2014, pp. 58–69.

[16] E. Chavez, G. Navarro, R. Baeza-Yates, J.L. Marroquin, Searching in metric spaces, ACM Comput. Surv. 33 (3) (2001) 273–321.

[17] K.L. Clarkson, Nearest neighbor queries in metric spaces, Discrete Comput. Geom. 22 (1999) 63–93.

[18] A. Esuli, Pp-index: using permutation prefixes for efficient and scalable approximate similarity search, in: Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR), 2009, pp. 17–24.

[19] A. Esuli, Use of permutation prefixes for efficient and scalable approximate similarity search, Inf. Process. Manag. 48 (5) (2012) 889–902.

[20] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Trans. Algorithms 3 (2) (2007). (article 20).

[21] K. Figueroa, K. Fredriksson, Speeding up permutation based indexing with indexing, In: Proceedings of the 2nd International Workshop on Similarity Search and Applications (SISAP), 2009, pp. 107–114.

[22] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, in: Proceedings of the

2nd European Conference on Computational Learning Theory (EuroCOLT), 1995, pp. 23–37.

[23] J. Gao, H.V. Jagadish, W. Lu, B.C. Ooi, DSH: Data sensitive hashing for high-dimensional k-nnsearch, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2014, pp. 1127–1138.

[24] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: Proceedings of the 25th International Conference on Very Large Data Bases (VLDB), 1999, pp. 518–529.

[25] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 368–373.

[26] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.

[27] D.A. Grossman, O. Frieder, Information Retrieval: Algorithms and Heuristics, Springer, Netherlands, 2004.

[28] G.R. Hjaltason, H. Samet, Index-driven similarity search in metric spaces, ACM Trans. Database Syst. 28 (4) (2003) 517–580.

[29] H. Hyyrö, Bit-parallel LCS-length computation revisited, in: 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004), 2004, pp. 16–27.

[30] P. Indyk, Handbook of Discrete and Computational Geometry, 2nd edition, CRC Press, MA, USA, 2004 (Chapter 39).

[31] D. Kirkpatrick, Optimal search in planar subdivisions, SIAM J. Comput. 12 (1983) 28–35.

[32] M. Kyselak, D. Novak, P. Zezula, Stabilizing the recall in similarity search, in: Proceedings of the 4th International Conference on Similarity Search and Applications (SISAP), 2011, pp. 43–49.

[33] D.-T. Lee, On *k*-nearest neighbor Voronoi diagrams in the plane, IEEE Trans. Comput. C-31 (6) (1982).

[34] Y. Malkov, A. Ponomarenko, A. Logvinov, V. Krylov, Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces, in: Proceedings of the 5th International Conference on Similarity Search and Applications (SISAP), 2012, pp. 132–147.

[35] Y. Malkov, A. Ponomarenko, A. Logvinov, V. Krylov, Approximate nearest neighbor algorithm based on navigable small world graphs, Inf. Syst. 45 (2014) 61–68.

[36] J.I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representations of permutations and functions, Theor. Comput. Sci. 438 (2012) 74–88.

[37] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88.

[38] G. Navarro, Searching in metric spaces by spatial approximation, VLDB J. 11 (1) (2002) 28–46.

[39] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007).

[40] M. Norouzi, A. Punjani, D.J. Fleet, Fast exact search in Hamming space with multi-index hashing, IEEE Trans. Pattern Anal. Mach. Intell. 36 (6) (2014) 1107–1119.

[41] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 60–70.

[42] M. Patella, P. Ciaccia, Approximate similarity search: a multi-faceted problem, J. Discrete Algorithms 7 (1) (2009) 36–48.

[43] V. Pestov, Intrinsic dimension of a dataset: what properties does one expect? in: Proceedings of the 20th International Joint Conference on Neural Networks, 2007, pp. 1775–1780.

[44] V. Pestov, An axiomatic approach to intrinsic dimension of a dataset, Neural Netw. 21 (2–3) (2008) 204–213.

[45] V. Pestov, Indexability, concentration, and VC theory, in: Proceedings of the 3rd International Conference on Similarity Search and Applications (SISAP), 2010, pp. 3–12.

[46] V. Pestov, Intrinsic dimensionality, ACM SIGSPATIAL 2 (2010) 8–11.

[47] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, CA, USA, 2006.

[48] U. Shaft, R. Ramakrishnan, Theory of nearest neighbors indexability, ACM Trans. Database Syst. 31 (2006) 814–838.

[49] T. Skopal, Unified framework for fast exact and approximate search in dissimilarity spaces, ACM Trans. Database Syst. 32 (4) (2007).

[50] T. Skopal, B. Bustos, On nonmetric similarity search problems in complex domains, ACM Comput. Surv. 43 (4) (2011). (article 34).

[51] E.S. Tellez, Practical Proximity Searching in Large Metric Databases (Ph.D. thesis), Universidad Michoacana de San Nicolás de Hidalgo, Morelia, Michoacán, Mexico, 2012.

[52] E.S. Tellez, E. Chavez, On locality sensitive hashing in metric spaces, in: Proceedings of the 3rd International Conference on Similarity Search and Applications (SISAP), 2010, pp. 67–74.

[53] E.S. Tellez, E. Chavez, A. Camarena-Ibarrola, A brief index for proximity searching, in: Proceedings of the 14th Iberoamerican Congress on Pattern Recognition (CIARP), 2009, pp. 529–536.

[54] E.S. Tellez, E. Chavez, G. Navarro, Succinct nearest neighbor search, Inf. Syst. 38 (7) (2013) 1019–1030.

[55] I. Volnyansky, V. Pestov, Curse of dimensionality in pivot based indexes, in: Proceedings of the 2nd International Workshop on Similarity Search and Applications (SISAP), 2009, pp. 39–46.

[56] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edition, Morgan Kaufmann, CA, USA, 1999.

[57] P. Zezula, G. Amato, V. Dohnal, M. Batko, Similarity search - the metric space approach, in: Database Systems, vol. 32, Springer, NY, USA, 2006.