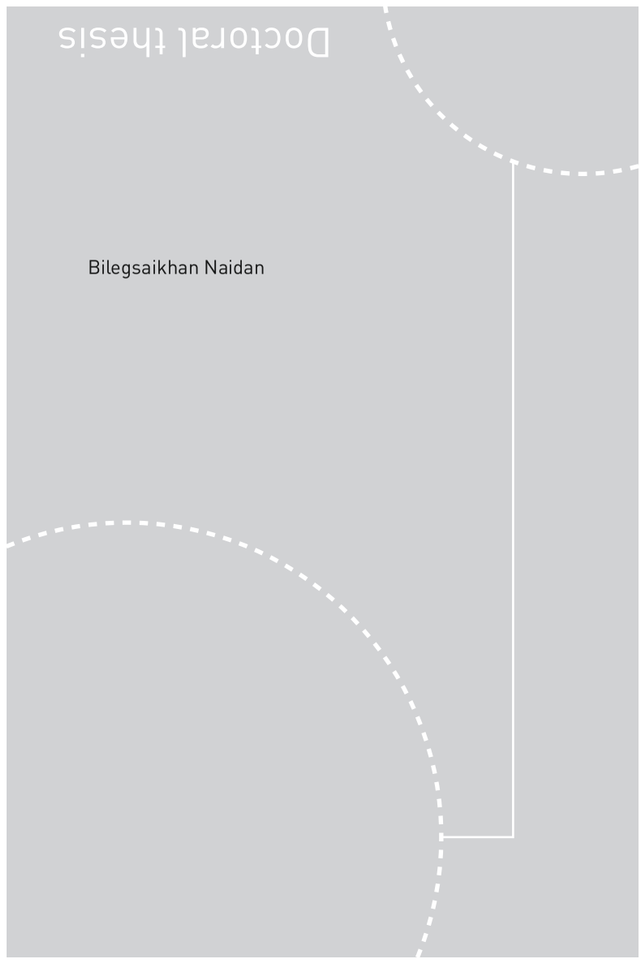
Doctoral theses at NTNU, 2017:84

Bilegsaikhan Naidan

Engineering Efficient Similarity Search Methods



ISBN 978-82-326-2240-5 (printed ver.)
ISBN 978-82-326-2241-2 (electronic ver.)
ISSN 1503-8181

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of
Philosophiae Doctor
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Doctoral theses at NTNU, 2017:84

NTNU
Norwegian University of
Science and Technology

Bilegsaikhan Naidan

Engineering Efficient Similarity Search Methods

Thesis for the Degree of Philosophiae Doctor

Trondheim, May 2017

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Department of Computer Science

ISBN 978-82-326-2240-5 (printed ver.)
ISBN 978-82-326-2241-2 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2017:84

Printed by NTNU Grafisk senter

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree of philosophiae doctor. This doctoral work has been performed at the Department of Computer and Information Science, NTNU, Trondheim, with Associate Professor Magnus Lie Hetland as main supervisor and with co-supervisors Professor Svein Erik Bratsberg and Professor Arne Halaas.

The thesis has been part of the research project Information Access Disruptions (iAd) project which is funded by the Norwegian Research Council. The main research of iAd project is to develop the next generation of search and the delivery technology in the information access domain.

Summary

Similarity search has become one of the important parts of many applications including multimedia retrieval, pattern recognition and machine learning. In this problem, the main task is to retrieve most similar (closest) objects in a large data set to a query efficiently by using a function that gives the distance between two objects.

This thesis is presented as a collection of seven papers with a tutorial on the topic and is mainly focused on efficiency issues of similarity search.

- Paper I provides a survey on state-of-the-art approximate methods for metric and non-metric spaces.
- Paper II proposes an approximate indexing method for the non-metric Bregman divergences.
- Paper III shows how to transform static indexing methods into dynamic ones.
- Paper IV presents a learning method to prune in metric and non-metric spaces.
- Paper V improves the existing approximate technique that can be applied to *ball*-based metric indexing methods.
- Paper VI presents an open source cross-platform similarity search library and a toolkit called the "Non-Metric Space Library" (NMSLIB) for evaluation of similarity search methods.
- Paper VII evaluates a metric index for approximate string matching.

Acknowledgements

First and foremost, I would like to thank my supervisor Magnus Lie Hetland. He has always been available to discuss research ideas when I needed and has been supporting me throughout this PhD work and helping me overcome a personal hard time. He also taught me how to conduct experiments and write scientific papers.

I would like to thank my supervisor Svein Erik Bratsberg for his support, research ideas and inspiring discussions.

I would like also thank Professor Arne Halaas for being co-supervisor.

Dr. Øystein Torbjørnsen has been an unofficial advisor to me. Despite being busy with his schedule, he led several fruitful meetings together with my supervisors when I struggled to produce publications.

I would like to give a big thank to Leonid Boytsov for his collaboration and co-authoring several papers and co-authoring our research Non-Metric Space Library and maintaining it. I would like to thank all people who contributed to the library.

I would like to thank Professor Eric Nyberg for his support of our library.

Finally, I would like to thank my family and they have been supporting me all time.

Contents

Part I

Introduction and Overview

Chapter 1

Introduction

Outline
This thesis is a collection of papers with a basic introduction on similarity search. This chapter presents the motivation and research goals of this work. Chapter 2 describes the theoretical background and a tutorial on similarity search. Chapter 3 summarizes the results and reviews the papers. It also discusses future work and concludes the thesis. The research contribution of this thesis is defined by seven research papers which can be found in Part II.

1.1 Motivation

Searching has become one of the crucial parts in our daily life and it allows us to find useful information. Traditional text-based search can not be directly applied to more complex data types (such as multimedia data) which do not have any text information (for instance, caption, description and tags). So-called similarity search can be applied to those data types. In similarity search, domain objects (such as strings and multimedia data) are modeled in a metric or non-metric space and those objects are retrieved based on their similarity to some given query.

For multimedia data, similarity search is often performed on the feature vectors extracted from the real data. Feature vectors are usually represented as high-dimensional data, so due to the so-called curse of dimensionality phenomenon, the efficiency of the indexing structures deteriorates rapidly as the number of dimensions increases. This may be because there is a trend for the objects to be

Some of the text and figures included in this thesis are based on my PhD plan which was partially included in a journal paper without my permission.

almost equidistant from the query objects in a high-dimensional space.

Let us consider two use cases of similarity search in two different domains.

Multimedia retrieval - In image retrieval, similar images to a query image are retrieved by using image features such as shape, edge, position and color histogram. So-called the Signature Quadratic Form Distance [5] (SQFD) is shown to be effective in image retrieval. In the method of Beecks [5], an image is mapped into a feature space and then its features are clustered with the standard k-means clustering algorithm. Each cluster is represented by a centroid and a cluster weight. Figure 1.1 shows three example images and the visualizations of their corresponding features as multiple circles. The centroid, weight and color of each cluster are represented by the center, diameter and color of a circle, respectively. We can see that the first two images are more similar to each other than the third one and that observation is also true for their feature signatures.

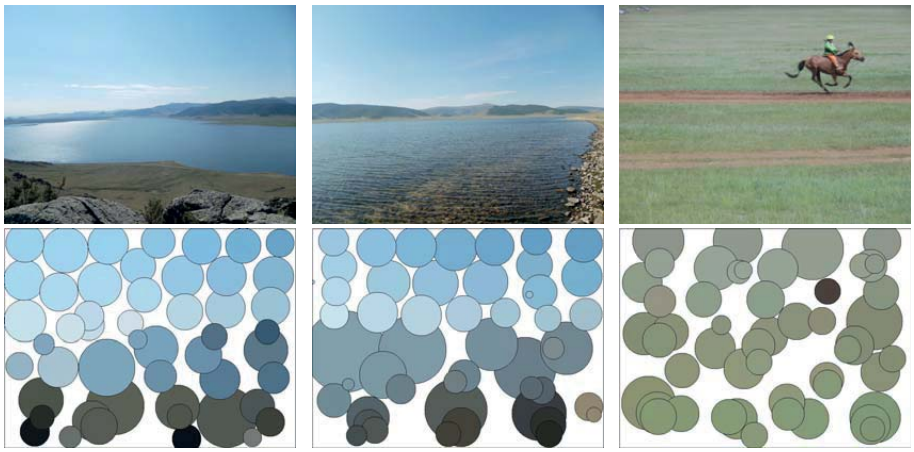


Figure 1.1: Three example images at top and the corresponding visualizations of their feature signatures at bottom.

Machine learning - In k-nearest neighbor classification, a test instance needs to be assigned into one of the several classes. First, we need to find the k closest training instances of the test instance. Then, the most occurred class is selected as the final decision. Figure 1.2 shows an example of k-nearest neighbor classification where $k = 3$. There are two classes of training examples which are marked as the blue and green points. The query instance (marked as the red point) is classified as the blue class since two points out of 3 nearest neighbors of the query are blue.

The term of *metric indexing structure* refers to data structures which are designed to store our objects and allow us to find similar objects quickly.

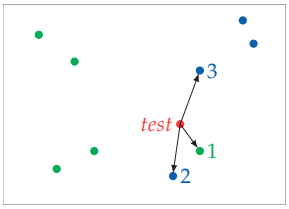


Figure 1.2: An example of k-nearest neighbor classification.

A naive approach to find similar objects in a data set to a query object is the linear scan. Essentially, it compares every object in the data set with the query. However, it is impractical for large data sets. Also, some of distance functions are very expensive to compute. For example, the distance computation of two text strings with length n (by using the edit distance with time complexity of $\Omega(n^2)$) is more expensive than the distance computation of two real valued vectors with dimension n (by using the Euclidean distance with time complexity of $O(n)$) in terms of CPU costs.

Therefore, numerous index structures have been proposed in the field of similarity search to reduce both of computational and I/O costs. Nowadays data sets tend to be rapidly growing in practice and thus the overall search costs would become unacceptable for those large data sets. A way to reduce the cost of similarity query processing for large data sets, approximate search (relevant objects that are missing from the result) and similarity search in parallel/distributed environment (using several processors/computers) are becoming increasingly important and popular.

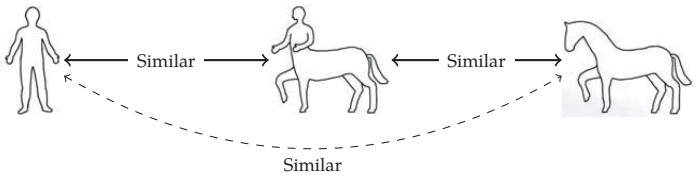


Figure 1.3: Illustration of the transitivity property of the triangle inequality.

The triangle inequality is the key component of metric index structures. It holds the transitivity property in a metric space. In Figure 1.3, we say that *the man is similar to the centaur* and *the centaur is similar to the horse*. Then, because of the transitivity property, the triangle inequality would suggest us that *the man is similar to the horse*, and vice versa [40]. But it does not match with our intuition. Therefore in some applications, our intuitions of similarity are better modeled with non-metric spaces [40]. However, we can not directly apply metric index structures to

non-metric spaces.

Another problem is that the existing metric index structures are static. That means that, once they are built for a given data set, adding (removing) more objects to (from) the index may result in full re-construction of the index which is usually time consuming and computationally intensive.

In this thesis, we addressed the problems mentioned above and mainly focused on approximate methods since exact methods are not efficient in high dimensional and non-metric spaces.

1.2 Research goals

Broadly speaking, the problem of similarity search can be divided into two main categories:

1. Figure out how to represent the complex objects of a data domain and find effective distance functions for that representations. In image retrieval, these two tasks could be to extract image features (such as cluster weights) and find distance functions (such as SQFD) that would retrieve most similar images using their features.
2. Design efficient similarity search methods for objects. In image retrieval, the task could be to find most similar images using their features quickly.

The main objective of this PhD project belongs to the second category. Three more specific research goals towards the main objective are defined as:

RG1. Propose new similarity search methods;

RG2. Improve and/or evaluate the efficiency of the existing similarity search methods;

RG3. Broaden the applicability of similarity search methods.

I will review these goals in Section 3.4 together with the papers. The first two goals aim at investigating the design and evaluation of efficient strategies for fast similarity searches while the third goal investigates how methods can be applied to various spaces. The efficiency of methods is measured in terms of the number of distance computations or CPU time required to process a query.

Chapter 2

Background

Outline
This chapter gives the basic theoretical knowledge relevant to understanding this thesis and the problem description for the similarity search based on the metric space model. First, we introduce the metric space model and a few examples of distance functions. Similarity queries and problem description are given in the two next sections. Then, a naive index structure, space partitioning principles and pruning principles are presented in the following sections. Finally, various types of methods are explained in the last several sections.

2.1 Metric space model

Definition 2.1. A metric space is a tuple $\langle \mathbb{U}, d \rangle$ of a set $\mathbb{U}$ and a function $d : \mathbb{U} \times \mathbb{U} \to \mathbb{R}$ which satisfies the following four properties for all $x, y, z \in \mathbb{U}$:

- $d(x, y) \geq 0$ (non-negativity);
- $d(x, y) = 0 \Leftrightarrow x = y$ (identity);
- $d(x, y) = d(y, x)$ (symmetry);
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

A distance function is called *metric* if it satisfies all of the four properties of a metric listed above. We call a distance function *semi-metric* if it satisfies the properties of non-negativity, identity and symmetry and does not satisfy the triangle inequality. A distance function that does not satisfy the properties of symmetry and triangle inequality notated as *non-metric*.

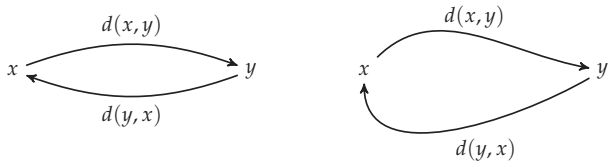


Figure 2.1: Illustrations of (a) metric and (b) non-metric distance functions

Illustrations of metric and non-metric distance functions are shown in Figure 2.1.

2.2 Distance functions

Distance functions are used to access the similarity between two objects. The smaller the distance between two objects, the more they similar. Distance functions are defined with respect to their target domains and applications. In the following, we provide a few examples of (dis)similarity measures.

The Minkowski distance - The distance between two n-dimensional vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ (for instance, color histograms) can be measured with the Minkowski distance which is defined as:

$$\mathcal{L}_p(x, y) = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

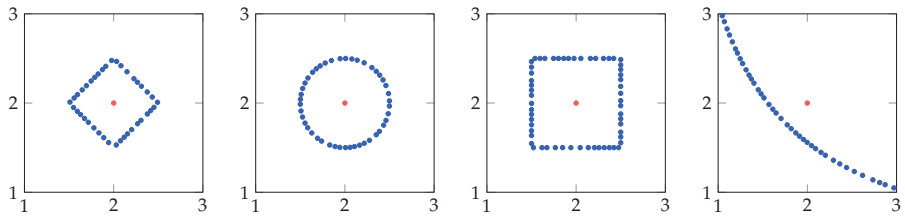
Some well-known special cases are $\mathcal{L}_1$ (Manhattan distance), $\mathcal{L}_2$ (Euclidean distance) and $\mathcal{L}_\infty$ (Chebyshev/maximum distance). However, the Chebyshev distance is defined with the following explicit formula:

$$\mathcal{L}_\infty(x, y) = \max_{1 \le i \le n} |x_i - y_i|$$

The time complexity of Minkowski distance is $O(n)$.

The Edit distance - The similarity of two strings can be measured by the edit distance (or Levenstein distance). This measure computes the minimal number of insertions, deletions, and replacements required to transform one string into another. The time complexity of edit distance is $\Omega(n^2)$.

The Hamming distance - The set of all 2^n binary strings of length n is called the Hamming space. The Hamming distance of two equal length binary strings is defined as the number of positions for which the bits are different.



(a) Manhattan (b) Euclidean (c) Chebyshev (d) KL-divergence

Figure 2.2: The blue points are at the distance value of 0.5 from the red point $(2, 2)$ with the corresponding distance functions.

The Kullback–Leibler (KL) divergence - The distance between two probability distributions $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ can be measured with the KL divergence which is defined as:

$$d(x, y) = \sum_{i=1}^{n} x_i \log \frac{x_i}{y_i}$$

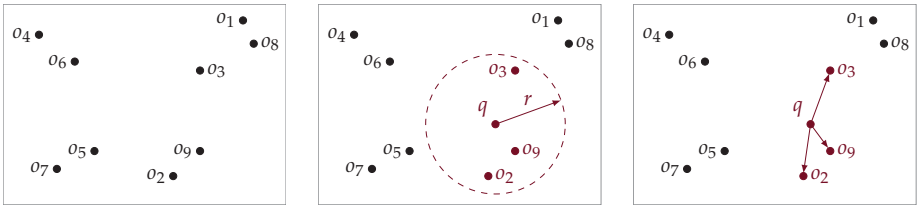
The KL divergence belongs to the family of non-metric distances so called Bregman divergences [10].

The Signature Quadratic Form distance (SQFD) [5] - This function is designed for measuring the similarity between two images. For two image features $S^x = \{\langle c_i^x, w_i^x \rangle | i = 1 \ldots n\}$ and $S^y = \{\langle c_i^y, w_i^y \rangle | i = 1 \ldots m\}$, the SQFD is calculated as:

$$d_f(S^x, S^y) = \sqrt{w \cdot M \cdot w^T}$$

where $w = (w_1^x, \ldots, w_n^x, -w_1^y, \ldots, -w_m^y)$ and $M \in \mathbb{R}^{(n+m)\times(n+m)}$ is the matrix, in which each element m_{ij} is obtained by applying a similarity function $f(c_i, c_j)$ on $c = (c_1^x, \ldots, c_n^x, c_1^y, \ldots, c_m^y)$. An example of $f(c_i, c_j)$ is $\frac{1}{1+\mathcal{L}_2(c_i, c_j)}$.

Illustrations of some of the distance functions discussed above are shown in Figure 2.2, where the blue points are at the same distance from the red point in 2-dimensional vector space.



(a) (b) (c)

Figure 2.3: Examples of (a) sample data points in 2-dimensional Euclidean space, (b) a range(q, r) query and (c) a k-NN$(q, 3)$ query.

2.3 Similarity queries

Several types of similarity queries have been proposed in the literature. We review the most common two of them. In order to ease the understanding of the concept and content of this chapter, we use nine sample points $\{o_1, \ldots, o_9\}$ in 2-dimensional Euclidean space which are shown in Figure 2.3a.

Range query - Given a query object $q \in \mathbb{U}$ and a maximum search distance r, it retrieves all objects from a data set $\mathbb{D}$ whose distance to q are equal to or less than r:

$$range(q, r) = \{x \in \mathbb{D} \mid d(x, q) \leq r\}$$

In the example given in Figure 2.3b, $\{o_2, o_3, o_9\}$ are reported as the result of the query.

k-Nearest Neighbor (k-NN) query - Given a query object $q \in \mathbb{U}$ and the number of requested objects k, it retrieves a set of k objects $\mathbb{K}$ from a data set $\mathbb{D}$ whose distances to q are not larger than the distance of any remaining objects in $\mathbb{D}$:

$$k\text{-}NN(q, k) = \{x \in \mathbb{K} \mid \mathbb{K} \subseteq \mathbb{D}, |\mathbb{K}| = k, \forall y \in \mathbb{D} \setminus \mathbb{K}, d(q, x) \leq d(q, y)\}$$

In the example given in Figure 2.3c, $\{o_9, o_2, o_3\}$ are reported as the result of the query.

We note that there are two types of queries if the distance function is not symmetric. We call *left* queries if an object compared to the query is the first argument of $d(x, y)$, while the query is the second argument.

2.4 Problem description

Let $\mathbb{D} \subset \mathbb{U}$ and $d(x, y)$ be our data set and distance function, respectively. Our task is to retrieve a small set of the most relevant objects (either all within a search radius r, or the k nearest neighbors) in $\mathbb{D}$ for queries drawn from $\mathbb{U}$ at low computational costs (such as the total number of distance computations at query time) as possible.

There are several important evaluation criteria for similarity search methods:

- The number of distance computations required to process a query;
- The quality measures of an approximate query, such as recall;
- CPU time and memory required to process a query;
- Applicability, such as only for vector space;
- Search modality, such as range, nearest neighbors and join queries;
- Scalability–that is, the search cost is independent of the size of data sets;
- Dynamicity–that is, support insert and delete operations of objects;
- Capability to be stored on and loaded from disk–that is, to deal with large data sets;
- The number of disk access required to process a query;
- Capability to run in parallel and distributed environments.

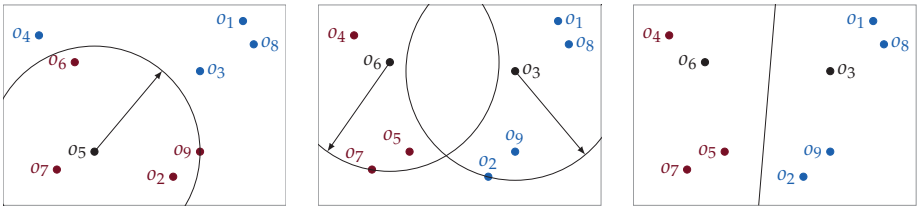
2.5 Space partitioning principles

One way to speed up similarity retrieval is to divide our data set into several disjoint subsets and expect that some subsets may be discarded during search. Because of the lack of a coordinate system in general metric spaces, the partitioning can be defined with help of some selected objects (so called pivots) from the data set. Let us consider the three basic metric partitioning principles for a subset $\mathbb{S} \subseteq \mathbb{D}$.

Ball partitioning principle - It divides $\mathbb{S}$ into two subsets $\mathbb{S}_1$ and $\mathbb{S}_2$ using a pivot $p \in \mathbb{S}$ and a spherical radius r. $\mathbb{S}_1$ contains a set of objects that are inside of the sphere (i.e. $\mathbb{S}_1 \leftarrow \{o_i \in \mathbb{S} \mid d(p, o_i) \leq r\}$) while $\mathbb{S}_2$ contains the remaining objects from $\mathbb{S}$ (i.e. $\mathbb{S}_2 = \mathbb{S} \setminus \mathbb{S}_1$).

In Figure 2.4a, we selected o_5 as the pivot and apply the ball partitioning on the sample data points given in Figure 2.3a to obtain two subsets $\mathbb{S}_1 = \{o_2, o_5, o_6, o_7, o_9\}$ and $\mathbb{S}_2 = \{o_1, o_3, o_4, o_8\}$.

Bisector partitioning principle - It divides $\mathbb{S}$ into two subsets $\mathbb{S}_1$ and $\mathbb{S}_2$ with respect to two pivots $\{p_1, p_2\} \in \mathbb{S}$. Thus, $\mathbb{S}_1$ consists of objects that are closer to p_1 than p_2 (i.e. $\mathbb{S}_1 \leftarrow \{o_i \in \mathbb{S} \mid d(p_1, o_i) \leq d(p_2, o_i)\}$) while $\mathbb{S}_2$ consists of the remaining objects from $\mathbb{S}$ (i.e. $\mathbb{S}_2 = \mathbb{S} \setminus \mathbb{S}_1$). Each partition has its own ball



(a) Ball (b) Bisector (c) Hyperplane

Figure 2.4: Illustrations of space partitioning principles

and the corresponding ball radius which is equal to the maximum distance between the pivot and all objects in that partition.

In Figure 2.4b, we select two pivots o_6 and o_3 and apply the bisector partitioning on the sample data points given in Figure 2.3a to obtain two subsets $\mathbb{S}_1 = \{o_4, o_5, o_6, o_7\}$ and $\mathbb{S}_2 = \{o_1, o_2, o_3, o_8, o_9\}$.

Generalized hyperplane partitioning - This partitioning principle is similar to the bisector partitioning, however, the hyperplane formed by two pivots is used in space pruning during search instead of balls. Figure 2.4c gives an illustration of the hyperplane partitioning principle.

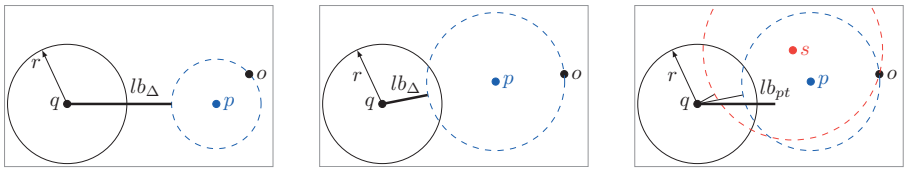
2.6 Pruning principles

The triangle inequality is employed as a fundamental property of existing metric index structures. The common usage of the triangle inequality is to estimate the lower bound $lb(q,o)$ of the actual distance $d(q,o)$ between an (indexed) object o in the index and a given query q using the precomputed distance $d(p,o)$ between the object and a pivot p. The lower bound $lb(q,o)$ of $d(q,o)$ based on the triangle inequality is computed as:

$$lb_\Delta(q,o) = |d(q,p) - d(p,o)|$$

Figure 2.5a and 2.5b show examples of the computation of the lower bound $lb_\Delta(q,o)$.

In general, if the lower bound of $d(q,o)$ is greater than a query radius r, the object can be safely discarded because $d(q,o) \geq lb(q,o) > r$. Otherwise, $d(q,o)$ needs to be computed and compared with the radius. Thus, the performance of an index structure directly depends on the number of objects that can be filtered out during search. This filtering power could be improved with several pivots by selecting the



(a) (b) (c)

Figure 2.5: Illustrations of (a-b) triangular lower bound $lb_\Delta(q,o)$ with pivot p and (c) ptolemaic lower bound $lb_p(q,o)$ with two pivots p and s.

maximum of lower bounds computed over those pivots. However, the computational cost for estimating lower bound with several pivots should be cheaper than the cost for computing the actual distance. Otherwise, this effort would become useless.

Recently, Hetland [35] has introduced *ptolemaic indexing* in which Ptolemy's inequality is applied to estimate the lower bounds of distances. For any four objects x, y, u, v, Ptolemy's inequality states:

$$d(x,v) \cdot d(y,u) \leq d(x,y) \cdot d(u,v) + d(x,u) \cdot d(y,v)$$

For a query q, object o, and pivots p and s in a set of pivots P, the lower bound $lb(q,o)$ of $d(q,o)$ based on Ptolemy's inequality is computed as:

$$lb_{pt}(q,o) = \max_{p,s \in P} \frac{|d(q,p) \cdot d(o,s) - d(q,s) \cdot d(o,p)|}{d(p,s)}$$

Ptolemy's inequality holds for quadratic form metrics including the well-known Euclidean distance. An example of lower bound $lb_p(q,o)$ is given in Figure 2.5c. We see that the triangle inequality does not always give optimal bounds (see Figure 2.5b) and that is heavily depend on pivot selection. In Figure 2.5c, the object o can not be discarded by using the two triangular lower bounds with pivots p and s because the query radius is larger than both bounds. However, in this case, the object o can be discarded with the ptolemaic bound. Ptolemaic indexing structures [36] have efficiently applied in content based image retrieval.

Let $d(p_1,o) \leq d(p_2,o)$. The triangular lower bound $lb(q,o)$ of $d(q,o)$ based on the hyperplane partitioning with two pivots p_1 and p_2 is computed as:

$$lb_\Delta(q,o) = \max\{(d(q,p_1) - d(q,p_2))/2, 0\}$$

Figure 2.6 shows an illustration of lower bound $lb_\Delta(q,o)$ of $d(q,o)$.

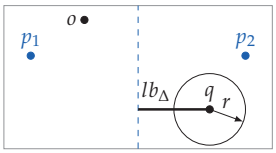


Figure 2.6: Illustration of triangular lower bound $lb_\Delta(q, o)$ of $d(q, o)$ with the hyperplane (blue dashed-vertical line) formed by p_1 and p_2.

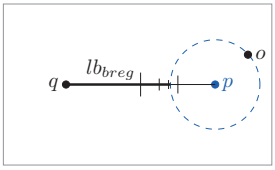


Figure 2.7: Illustration of Bregman lower bound $lb_{breg}(q, o)$ of $d(q, o)$.

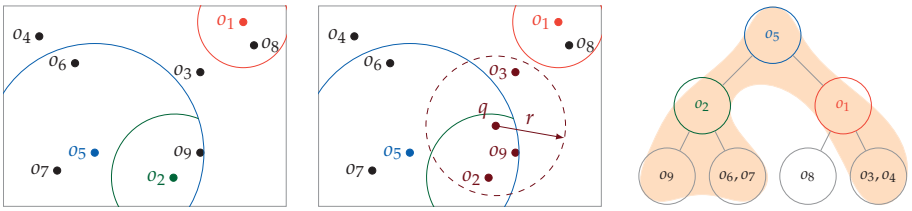
Non-metric spaces are neither symmetric nor triangular and thus it is impossible to apply the triangle inequality to estimate lower bounds for those spaces. Thus, Cayton [13] showed that lower and upper bounds for Bregman balls can be computed with a binary search. Figure 2.7 shows an example of Bregman lower bound calculation. Four small vertical lines show that how the binary search can be done to find the lower bound.

2.7 Space decomposition methods

Most of coordinate-based spatial [30, 33] and metric trees [9, 11, 21, 54, 57] belong to this category. The main idea behind space decomposition methods is to divide the data set recursively into either overlapping or non-overlapping hierarchical regions. In these methods, the search algorithm often uses a best-first branch-and-bound approach with pruning of irrelevant regions.

Let us have a look at two simple hierarchical index structures based on ball and hyperplane partitioning principles.

The vantage point tree [57] (VP-tree) is a balanced binary tree based on ball partitioning principle. The VP-tree divides $\mathbb{S}$ into two subsets $\mathbb{S}_1$ and $\mathbb{S}_2$ according to ball partitioning, where r is the median of distances between p and objects from $\mathbb{S}$. Starting with $\mathbb{S} = \mathbb{D}$, this method recursively selects a pivot, applies ball partitioning, and builds left and right subtrees for $\mathbb{S}_1$ and $\mathbb{S}_2$ respectively if the size of the subsets is greater than an user-defined parameter. The $range(q, r)$ search algorithm



(a) (b) (c)

Figure 2.8: A VP-tree: (a) an illustration of hierarchical space partitions (b) an illustration of a range query q with radius r (c) the corresponding tree (the highlighted yellow regions are visited during the search).

recursively traverses the tree from the root to leaves. For a visited internal node with p, it evaluates the distance $d(q, p)$. The pivot p is reported if $d(q, p) \leq r$. The algorithm decides which subtrees to visit by defining lower bounds on the distances from q to objects in the left and right subtrees. It is necessary to traverse the left subtree only if $d(q, p) + r < m$, and, similarly, the right subtree only if $d(q, p) - r > m$. If none of these conditions meet, it needs to traverse both subtrees. For leaf nodes, the search sequentially computes the distances between q and all objects in the leaf node and reports qualified objects. We note that the main problem with the ball partitioning is that it is almost impossible to partition space without ball overlaps especially in high-dimensional spaces.

Figure 2.8 shows an example of a VP-tree in 2-dimensional Euclidean space. The algorithm for a range query q with radius r starts from the left subtree with pivot o_2 of the node associated with o_5 (because q is inside that region). We include o_2 in the result set of the query. Then, it visits the left leaf node of the node o_2. We also report o_9. Then, it needs to do backtracking to the node o_2 and visits the right leaf node that contains o_6, o_7. After that, it goes back to the node o_2 and the root node o_5. This time it visits the right subtree o_1 of the root node. The algorithm prunes the left leaf node of the node o_1 (because the query ball does not intersect with that region). Then, it visits the right leaf and finally, we report o_3.

The generalized hyperplane tree [54] (GH-tree) is also a binary tree and built similar to VP-trees. The GH-tree selects two pivots and applies generalized hyperplane partitioning principle in every recursive call of the tree building. We note that the resulting tree is not necessary to be balanced. The *range*(q, r) search algorithm for GH-trees is also similar to that of VP-trees. The left subtree is visited if $(d(q, p_1) - d(q, p_2))/2 \leq r$. And similarly, the right subtree is visited if $(d(q, p_2) - d(q, p_1))/2 \leq r$. We note that it is possible to visit both subtrees. Figure 2.9 shows an example of a GH-tree in 2-dimensional Euclidean space.

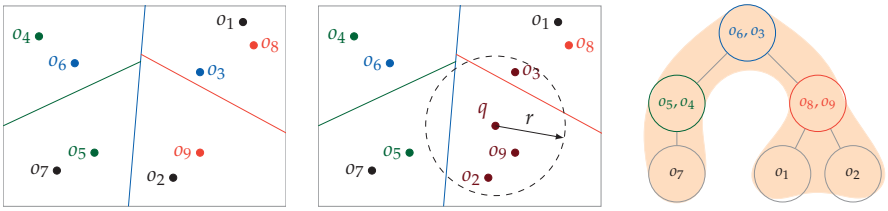


Figure 2.9: A GH-tree: (a) an illustration of hierarchical space partitions (b) an illustration of a range query q with radius r (c) the corresponding tree (the highlighted yellow regions are visited during the search).

2.8 Approximate methods

Exact similarity retrieval is inefficient in high-dimensional spaces due to the phenomenon known as the *curse of dimensionality*. Chavez et al. [16] generalized the curse of dimensionality in general metric spaces without coordinates and estimated the *intrinsic dimensionality* of $\mathbb{D}$ as, $\rho = \mu^2/(2\sigma^2)$, where μ and σ^2 are the mean and variance of distance distribution in $\mathbb{D}$. The intrinsic dimensionality is high if objects are placed almost equidistant from each other (the variance becomes low with respect to mean distance).

Figure 2.10 shows the intrinsic dimensionalities of two sample data sets. For the data set with high intrinsic dimensionality $\rho = 38.91$, distances are very concentrated around distance of 0.9.

Many of the space decomposition methods mentioned in Section 2.7 suffer from the spaces with high intrinsic dimensionality. Because most of the regions would be intersected with the query region. In worst case, these methods would perform more or less same as the linear scan. An example of this behaviour is shown in Figure 2.11a. We see that the performance of VP-trees degrades when the dimensionality increases beyond 10. As shown in Figure 2.11b, the intrinsic dimensionality increases as the dimensionality increases.

Therefore, approximate similarity retrieval is becoming a new trend to handle this problem, in which search results are relaxed by permitting errors while increasing the query performance.

Many of existing methods for approximate nearest neighbor search specifically focus on ϵ-nearest neighbor queries.

Definition 2.2. A ϵ-nearest neighbor (ϵ-NN) retrieves an object x' from $\mathbb{D}$ whose distance to q is not larger than the distance between the actual nearest neighbor x

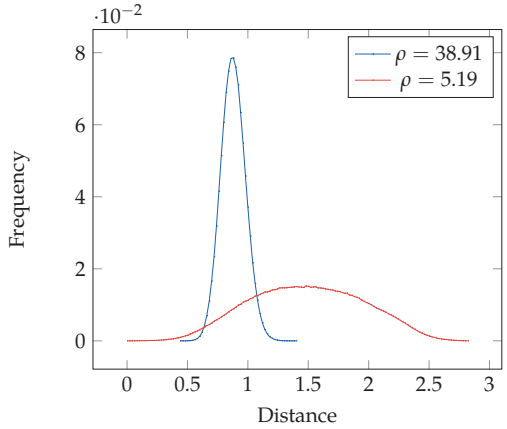
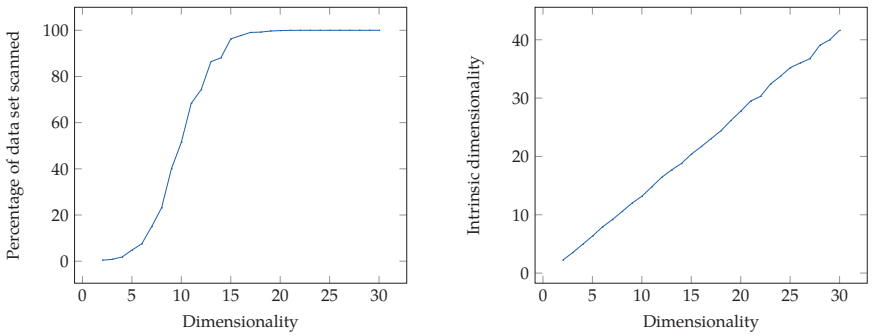


Figure 2.10: Distance distributions of two data sets with low ($\rho = 5.19$) and high ($\rho = 38.91$) intrinsic dimensionalities.



(a) (b)

Figure 2.11: Uniformly generated vectors with varying dimensionalities (a) NN-search performance of VP-tree (b) Intrinsic dimensionality vs. dimensionality

and q times $(1+\epsilon)$, where $\epsilon > 0$ (see Figure 2.12).

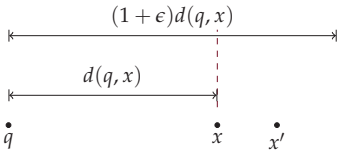


Figure 2.12: Example of an ϵ-NN. x' is ϵ-NN of x.

In [4, 17, 20, 59], the search algorithm uses $r/(1+\epsilon)$ instead of the real query radius r. It potentially would reduce the number of regions that intersect with the query region and therefore that leads speeding-up approximate query processing.

Another interesting type of approximate method is the early stopping technique which is used by Liu et al. [41] and Cayton [13]. The search algorithm is a best-first traversal in the tree and visits at most a specific number (a user-defined parameter) of leaf nodes of the tree. The main idea behind this method is that the search algorithm can find an object close to the actual nearest neighbor quickly.

Zezula et al. [59] proposed an early stopping technique based on distance distribution. The cumulative distance distribution $F_p(x)$ represents the set of objects $o_i \in \mathbb{D}$ for which $d(p,o_i) \leq x$. $F_q(x)$ is unknown in advance and we assume that query objects follow the same distribution as objects in $\mathbb{D}$. Therefore the search algorithm uses the global cumulative distance distribution $F(x)$ that is calculated for the entire data set. For a user-defined parameter t, the search algorithm stops if $F(d(q,o_k)) \leq t$, where o_k is the current k-NN found so far.

Amato et al. [1] introduced a pruning principle for metric balls based on distance distribution. For a metric ball containing a set of objects o_i, the search algorithm estimates the probability pr such that $d(q,o_i) \leq r$ and discards that ball if $pr \leq t$, where t is a user-defined parameter.

The TriGen method [51] transforms a semi-metric distance function into a metric one by using a modifier function $f(\cdot)$. Since the method uses a modified version of binary search algorithm, this modifier function should be concave or convex. In its essence, the TriGen method applies various modifiers on a sample data that drawn from the data set to optimize the best modifier that gives the minimum intrinsic dimensionality for a given user-defined error threshold. The error is calculated as the ratio between the number of triplets (i.e., $f(d(x,y))$, $f(d(y,z))$ and $f(d(x,z))$) that violate the triangle inequality and the number of all examined triplets. A concave function increases the intrinsic dimensionality because the difference between all modified distances becomes almost same whereas a convex function decreases the intrinsic dimensionality. A simple modifier that can be

used in the TriGen method is the Fractional-Power base (FP-base) and it is defined as:

$$f(d,w) = \begin{cases} d^{\frac{1}{1+w}}, & \text{for } w > 0 \\ d^{1-w}, & \text{otherwise.} \end{cases}$$

An example of FP-base is shown in Figure 2.13. Once we have the proper modifier, we can use an existing metric indexing structures.

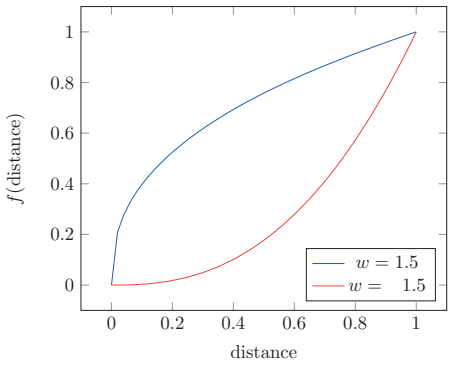
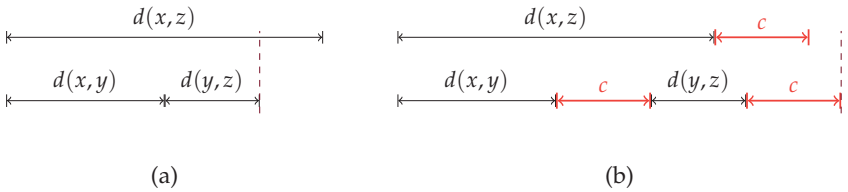


Figure 2.13: Illustration of the FP-base with $w = 1.5$ and $w = -1.5$.

Roth et al. [49] presented so-called the constant shift embedding method that converts a semi-metric distance function into a metric one by adding a positive constant to the calculated distance value (see Figure 2.14). Lei Chen et al. [19] proposed a method that relies on the constant shift embedding and supports both of exact and approximate queries.



(a) (b)

Figure 2.14: Constant shift embedding method: (a) $d(x,y) + d(y,z) < d(x,z)$ (b) $d(x,y) + d(y,z) + c > d(x,z)$ after adding a positive constant c to the distances.

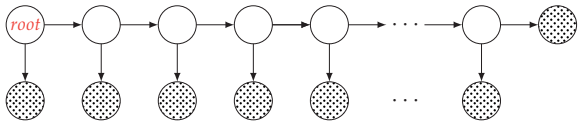


Figure 2.15: A list of clusters: leaf nodes are illustrated by dotted circles.

2.9 Methods based on clustering principle

The Sparse Spatial Selection tree [12] is a hierarchical unbalanced multiway tree. The first object is selected as the center of the first cluster center. Each object in the remaining set is selected as a new cluster center if the distance from the object to its nearest center is greater than αM, where M is the maximum distance between any two objects in the current set and α is a user-defined parameter. If the object is not selected as a cluster center then the object is added to the bucket of the cluster associated with its nearest center. The process is recursively applied to those clusters if they contains more than a user-defined number of objects.

List of clusters [15] is an unbalanced VP-tree in which each left node has a bucket which contains a certain number (a user-defined parameter) of objects whereas the right nodes contain the remaining objects. The process is recursively applied to the right node until the size of the node is below the parameter (Figure 2.15). The search algorithm for list of clusters is similar to one in VP-trees.

The bregman ball tree [13] is a hierarchical binary tree which is designed to solve nearest neighbor queries under Bregman divergences. The k-means algorithm is employed as the space partitioning method. It yields two clusters at each internal node and the resulting two clusters become the left and right nodes of the internal node.

Another multiway tree that relies on the k-means clustering algorithm is the hierarchical k-means tree [31] and its variant [44].

2.10 Projection methods

Projection methods are often based on a filter-and-refine principle. The basic idea behind these methods are quite simple. Instead of performing search in the original space, objects are transformed into the target space. Then, the search is performed in much cheaper target space to obtain a set of candidate objects. The candidate objects are refined in the original space. Figure 2.16 shows an illustration of projection method.

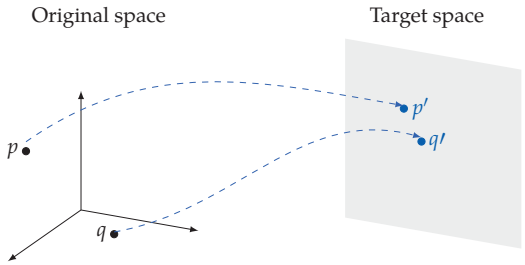


Figure 2.16: Projection of objects from original to target space.

Locality Sensitive Hashing [38, 39] (LSH) is an approximate method which places data points so that similar points end up in the *same* bucket with *high* probability. A hash family $\mathcal{H}(r, \epsilon, P_1, P_2)$ is called locality-sensitive if it satisfies the following two properties for any $x, y \in \mathbb{R}^d$:

- If x and y are close to each other ($d(x, y) \leq r$) then they should be end up in the same bucket with a high probability greater than or equal to P_1: $Pr_{\mathcal{H}}[h(x) = h(y)] \geq P_1$;
- If x and y are far apart from each other ($d(x, y) \geq (1 + \epsilon)r$) then they should be end up in the same bucket with a low probability $P_2 < P_1$: $Pr_{\mathcal{H}}[h(x) = h(y)] \leq P_2$.

For a given locality-sensitive hash family, approximate nearest neighbor queries can be solved with a hashing method. The construction algorithm for the LSH first builds l hash tables. Each hash table $\mathcal{T}_i$ is associated with a hash function $g_i(\cdot)$ which is defined by concatenating k randomly chosen hash functions h from H, i.e., $g_i(x) = [h_1(x), h_2(x), \ldots, h_k(x)]$. In other words, $g_i(x)$ represents the bucket of $\mathcal{T}_i$ which contains x. Then for each $\mathcal{T}_i$, the algorithm hashes all points from the data set into $\mathcal{T}_i$ (see Figure 2.17). To perform a query q, the algorithm hashes the query to find its corresponding buckets $\{g_1(q), g_2(q), \ldots, g_l(q)\}$. All points in those buckets form a set of candidate points which are eventually refined with the original distance function.

Approximate nearest neighbor search under the cosine distance between vectors can be solved with so-called the random projection method [3] of LSH. In this method, the hash function is defined as:

$$h(x) = \begin{cases} 1, & \text{if } v^T \cdot x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

where v is a random vector. For each g_i, all vectors are approximated by k dimensional bit vectors.

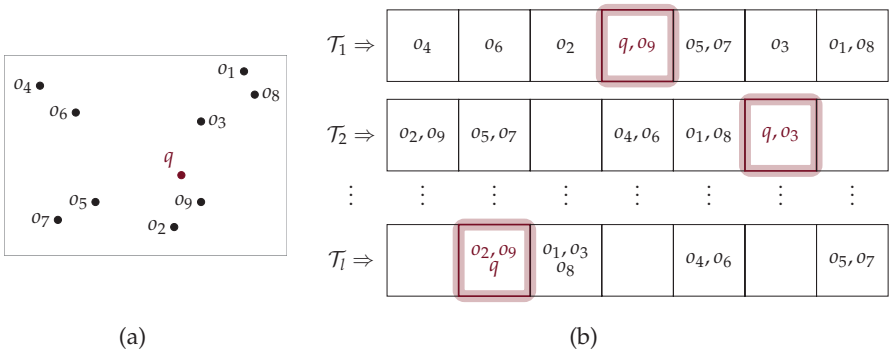
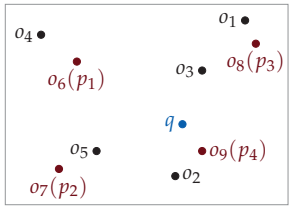


Figure 2.17: (a) Sample data and query points (b) Illustration of a LSH index: the squares are the buckets of the hash tables $\mathcal{T}_i$ and the points $\{o_9, o_3, \ldots, o_2\}$ in the highlighted buckets are selected as the candidate points.

Amato et al. [2] and Chávez et al. [32] proposed independently so-called permutation methods for approximate nearest neighbor search. In these methods, objects are projected from the original space to the target permutation space. We select m pivots at random from the data set. Then, each object is represented by a ranked list of pivots (permutations) order by the distance to the object. The search algorithm is based on an assumption that similar objects should have similar permutations.

We explain the permutation method of Chávez et al. [32]. An example of this method is shown in Figures 2.18. We select four objects o_6, o_5, o_8, o_9 and mark them as four pivots p_1, p_2, p_3, p_4, respectively (see Figure 2.18a). For each object, we compute the distances between the pivots and that object. Then, we order the pivots by increasing distance to the object. For object o_5, its first closest pivot is p_2. Similarly, its second, third, and forth closest pivots are p_1, p_4, and p_3, respectively. Thus the permutation for o_5 is (p_2, p_1, p_4, p_3). Let us denote by $perm(x)$ the permutation of a object x. Also, let $pivot\_pos_x(p_i)$ denote the position of a pivot p_i in a $perm(x)$. The position of p_1 in the permutation (p_2, p_1, p_4, p_3) is 2. Similarly, the positions of p_2, p_3, and p_4 are $1, 4$, and 3, respectively. Then the pivot position $pivot\_pos_{o_5}$ of the permutation induced by o_5 are $(2, 1, 4, 3)$. We repeat the same steps for other objects to obtain their permutations and corresponding pivot positions. We can see that two nearest neighbors of o_1 are o_8 and o_3. Their permutations and pivot positions are similar (see Figure 2.18b).

The search algorithm consists of two steps. In the filtering step, the pivot positions of objects are compared with the pivot position of query using a rank correlation metric. The rank correlation metrics include Kendall Tau, Spearman Footrule (equal to $\mathcal{L}_1$) and Spearman Rho (equal to squared $\mathcal{L}_2$). Brute-force filtering of



(a)

Object	Permutation (perm_x)	pivot_pos_x
o_1	(p_3, p_4, p_1, p_2)	$(3,4,1,2)$
o_2	(p_4, p_2, p_1, p_3)	$(3,2,4,1)$
o_3	(p_3, p_4, p_1, p_2)	$(3,4,1,2)$
o_4	(p_1, p_2, p_4, p_3)	$(1,2,4,3)$
o_5	(p_2, p_1, p_4, p_3)	$(2,1,4,3)$
o_6	(p_1, p_2, p_4, p_3)	$(1,2,4,3)$
o_7	(p_2, p_1, p_4, p_3)	$(2,1,4,3)$
o_8	(p_3, p_4, p_1, p_2)	$(3,4,1,2)$
o_9	(p_4, p_3, p_2, p_1)	$(4,3,2,1)$
q	(p_4, p_3, p_1, p_2)	$(3,4,2,1)$

Position of p_2 is 1

Similar

(b)

Object	Footrule($\text{pivot\_pos}_q, \text{pivot\_pos}_{o_i}$)
o_1	$\lvert 3-3\rvert + \lvert 4-4\rvert + \lvert 2-1\rvert + \lvert 1-2\rvert = 2$
o_3	$\lvert 3-3\rvert + \lvert 4-4\rvert + \lvert 2-1\rvert + \lvert 1-2\rvert = 2$
o_8	$\lvert 3-3\rvert + \lvert 4-4\rvert + \lvert 2-1\rvert + \lvert 1-2\rvert = 2$
o_9	$\lvert 3-4\rvert + \lvert 4-3\rvert + \lvert 2-2\rvert + \lvert 1-1\rvert = 2$
o_2	$\lvert 3-3\rvert + \lvert 4-2\rvert + \lvert 2-4\rvert + \lvert 1-1\rvert = 4$
o_4	$\lvert 3-1\rvert + \lvert 4-2\rvert + \lvert 2-4\rvert + \lvert 1-3\rvert = 8$
o_5	$\lvert 3-2\rvert + \lvert 4-1\rvert + \lvert 2-4\rvert + \lvert 1-3\rvert = 8$
o_6	$\lvert 3-1\rvert + \lvert 4-2\rvert + \lvert 2-4\rvert + \lvert 1-3\rvert = 8$
o_7	$\lvert 3-2\rvert + \lvert 4-1\rvert + \lvert 2-4\rvert + \lvert 1-3\rvert = 8$

Candidate points

(c)

Figure 2.18: An illustration of the permutation method of Chávez et al. [32] (a) pivot selection (b) permutation projection (c) calculation of the Spearman Footrule on permutations.

permutations and filtering based on incremental sorting are implemented in [32]. Objects with similar permutations to the permutation of query are selected as a set of candidate objects. In the refinement step, the candidate objects are compared with the query by computing the distance in the original space.

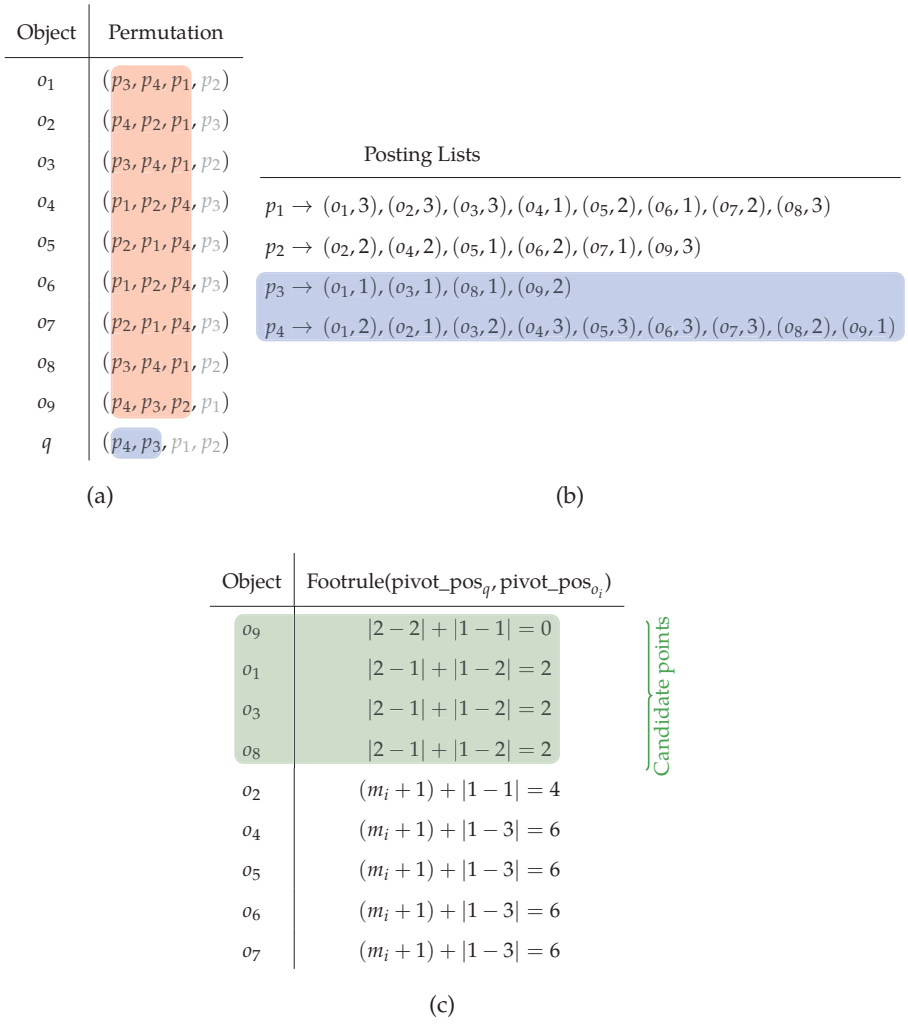
Figueroa et al. [28] used existing methods for metric spaces to index permutations.

We consider so-called the Metric Inverted File (MI-File) of Amato et al. [2]. The construction of permutations in MI-File is performed in the same manner as in the permutation index, but the most closest m_i ($m_i \leq m$) and m_s ($m_s \leq m_i$) pivots are used for indexing and searching, respectively. As we decrease m_i the index will be smaller whereas the search becomes faster if we decrease m_s. Each element of posting list is a pair $(x, pivot\_pos_x(p_i))$. The authors proposed several strategies for indexing and searching, but we explain one of the simple versions. In example given in Figure 2.19, we use $m_i = 3$ and $m_s = 2$. When performing a query, we only need to access to the posting lists associated with m_s closest pivots of $perm_q$. That is p_4 and p_3 for our example. We note that $pivot\_pos_q(p_3) = 2$ and $pivot\_pos_q(p_4) = 1$. When calculating the Spearman Footrule metric, we add $m_i + 1$ to the distance if the pivot position of a x is absent in the current processing posting list (because all pivots may not used during the search).

Tellez et al. [53] proposed a variant of the MI-file in which most closest m_i pivots to objects are used for indexing. Posting lists contain only object identifiers. Therefore, it is impossible to use a rank-correlation metric during the search. Thus, objects whose permutations have more common pivots with the permutation of the query are selected as candidate objects. The set intersection algorithm is used for this purpose.

Pivots are indexed with unique numbers ranging from 1 to m. Thus one can treat that permutations are strings over some pivot alphabet. Permutations with most closest $l \leq m$ pivots are used to build so-called the Permutation Prefix Index [25] (PP-Index) (see Figure 2.20). When searching the index with the permutation prefix of length $l_s \leq l$ of a query, we first find the subtree which shares a common prefix of length l_s with perm_q. If that subtree contains at least γ (a user-defined parameter) objects then those objects are selected as candidates. Otherwise, we repeat the same procedure with a shorter length $l_s = l_s - 1$. Let γ be 2 in our example. There is no such subtree with the prefix of 431. So we have to try again with the shorter prefix 43. This time, we obtain only o_9. Thus, we repeat again the same steps for the prefix 4 to obtain o_2 and o_9.

Another type of mapping of points in high-dimensional spaces into low-dimensional spaces is proposed in FastMap [27] and MetricMap [55] so that the distance between points are preserved approximately. In FastMap, all points are perpendicularly projected on k lines where k is the dimensionality of the target space and each line is formed by two pivots. For each point, k projected coordinate values



Object	Permutation
o_1	(p_3, p_4, p_1, p_2)
o_2	(p_4, p_2, p_1, p_3)
o_3	(p_3, p_4, p_1, p_2)
o_4	(p_1, p_2, p_4, p_3)
o_5	(p_2, p_1, p_4, p_3)
o_6	(p_1, p_2, p_4, p_3)
o_7	(p_2, p_1, p_4, p_3)
o_8	(p_3, p_4, p_1, p_2)
o_9	(p_4, p_3, p_2, p_1)
q	(p_4, p_3, p_1, p_2)

(a)

Posting Lists

$p_1 \rightarrow (o_1,3), (o_2,3), (o_3,3), (o_4,1), (o_5,2), (o_6,1), (o_7,2), (o_8,3)$

$p_2 \rightarrow (o_2,2), (o_4,2), (o_5,1), (o_6,2), (o_7,1), (o_9,3)$

$p_3 \rightarrow (o_1,1), (o_3,1), (o_8,1), (o_9,2)$

$p_4 \rightarrow (o_1,2), (o_2,1), (o_3,2), (o_4,3), (o_5,3), (o_6,3), (o_7,3), (o_8,2), (o_9,1)$

(b)

Object	Footrule(pivot_pos$_q$, pivot_pos$_{o_i}$)
o_9	$\lvert 2-2\rvert + \lvert 1-1\rvert = 0$
o_1	$\lvert 2-1\rvert + \lvert 1-2\rvert = 2$
o_3	$\lvert 2-1\rvert + \lvert 1-2\rvert = 2$
o_8	$\lvert 2-1\rvert + \lvert 1-2\rvert = 2$
o_2	$(m_i+1) + \lvert 1-1\rvert = 4$
o_4	$(m_i+1) + \lvert 1-3\rvert = 6$
o_5	$(m_i+1) + \lvert 1-3\rvert = 6$
o_6	$(m_i+1) + \lvert 1-3\rvert = 6$
o_7	$(m_i+1) + \lvert 1-3\rvert = 6$

Candidate points

(c)

Figure 2.19: A Metric Inverted File (a) permutation projection and the pivots in the orange area are used for indexing ($m_i = 3$) (b) illustration of its corresponding inverted index (c) calculation of the Spearman Footrule on two posting lists associated with $p3$ and $p4$ ($m_s = 2$).

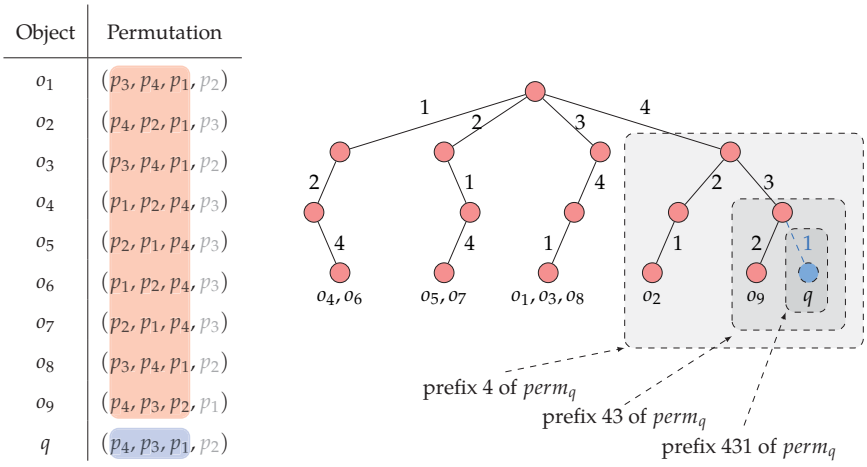
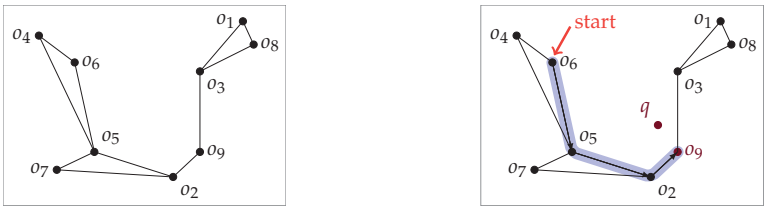


Figure 2.20: A Permutation Prefix Index: (a) permutation projection and the prefixes of length $l = 3$ are used for indexing (b) its corresponding prefix index and three shaded rectangles represent how the search is performed.

represent that point in the target space. We note that distance computation in low-dimensional spaces is less expensive than one in high-dimensional spaces.

2.11 Graph methods

The basic idea behind k-NN graphs is to build a graph in which each node is associated with exactly one object and is connected with its k nearest neighbors (see Figure 2.21a). The search algorithm is based on a greedy search which starts from a random node and moves to another node p that is closer to the query than the current node. The algorithm stops when such node p is not found (see Figure 2.21b). In order to improve the result, multiple searches could be done.



(a) (b)

Figure 2.21: A k-NN graph: (a) each object is connected with its two nearest neighbors (b) how the nearest neighbor query is performed.

The construction of an exact k-NN graph requires $n(n-1)/2$ pairwise distance computations which is unacceptable for large n. Therefore, there have been proposed many approximation methods for k-NN graph construction problem.

Dong et al. [23] proposed an approximate k-NN graph construction method in which each node is initially connected with random neighbors and then that node's neighbors are iteratively improved with the neighbors of the node's neighbors. This iterative algorithm is called neighborhood propagation.

A k-NN graph construction method based on the divide-and-conquer principle is introduced by Zhang et al. [60]. The LSH method is used to partition objects into several subsets. For each subset, the algorithm builds a k-NN graph using the brute force method. Then all resulting graphs are combined into a final graph.

So-called method SW-graph for approximating k-NN graph is proposed by Malkov et al. [42]. The graph construction algorithm relies on the greedy search algorithm which is described at the beginning of the section. All objects are added to the

graph one by one. For a new object, we create a node and find its nearest neighbor nodes in the graph. Then, we add undirected edges between the new node and the already found nearest neighbors.

2.12 Other methods

In the previous sections, I gave a brief survey of principles of metric-based indexing and approximation techniques (see tutorial [34], survey [18] and books [37, 58] for more details about these methods).

Another type of methods that exploits the parallel and distributed computing platforms has been reported in the literature. Cayton [14] proposed a method that utilizes multicore CPU and GPU platforms. So-called methods M-Chord [47] and MCAN [26] exploit the parallel query processing capabilities of distributed architectures.

Chapter 3

Research Summary

Outline
This chapter briefly summarizes the research underlying this thesis. First, Section 3.1 discusses the research method used in my research. Section 3.2 gives a list of the included papers, research process and the roles of the different authors. Section 3.4 evaluates our contributions. Possible future work is discussed in Section 3.6. Finally, Section 3.7 concludes the thesis.

3.1 Research method

During the last decades, researchers have been paid more attention to the practical implementations and evaluations of algorithms. Because there are some real-world indicators (such as cache effects and memory latency) that would influence the performance of algorithms and could be difficult to analyze theoretically [22]. So-called empirical algorithmics is a scientific methodology of studying the performance and behaviour of algorithms and data structures empirically. In this methodology [43], we need to define the following steps:

- Setup–that is, formulation of hypotheses, selection of data sets and programming language, etc.;
- Measures of performance, such as running time and the number of calls to an important subroutine[1];
- Analysis and interpretation of results.

[1] In similarity search, this could be the number of distance computations.

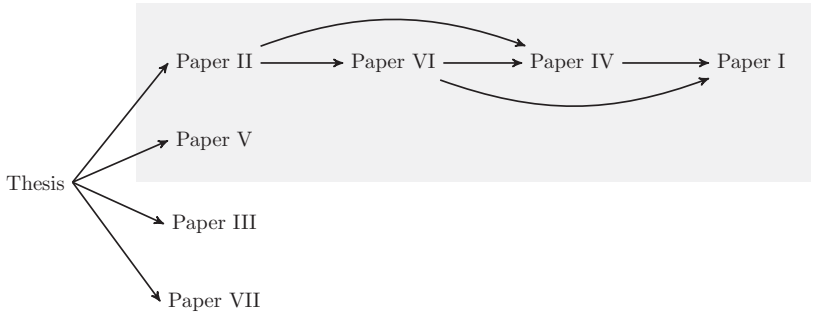


Figure 3.1: Relationship of the papers. The papers in the shaded rectangle address approximate similarity search.

Practical evaluation of the performance of a method also helps us to figure out bottlenecks and design more efficient methods.

3.2 Included papers

This section describes seven papers that are included in the thesis. The actual papers can be found in Part II. Figure 3.1 shows how the papers relate with each other.

3.2.1 Paper I

Bilegsaikhan Naidan, Leonid Boytsov and Eric Nyberg, Permutation Search Methods are Efficient, Yet Faster Search is Possible. VLDB 2015.

Abstract

We survey permutation-based methods for approximate *k*-nearest neighbor search. In these methods, every data point is represented by a ranked list of pivots sorted by the distance to this point. Such ranked lists are called permutations. The underpinning assumption is that, for both metric and non-metric spaces, the distance between permutations is a good proxy for the distance between original points. Thus, it should be possible to efficiently retrieve most true nearest neighbors by examining only a tiny subset of data points whose permutations are similar to the permutation of a query. We further test this assumption by carrying out an extensive experimental evaluation where permutation methods are pitted against state-of-the art benchmarks (the multi-probe LSH, the VP-tree, and proximity-graph based retrieval) on a variety of realistically large data set from the image and

textual domain. The focus is on the high-accuracy retrieval methods for generic spaces. Additionally, we assume that both data and indices are stored in main memory. We find permutation methods to be reasonably efficient and describe a setup where these methods are most useful. To ease reproducibility, we make our software and data sets publicly available.

Research process and roles of the authors

For my personal research projects, I developed a variety of search methods for generic spaces, which are called permutations methods. These methods were used as benchmarks in our other papers, however, our data sets were not sufficiently diverse. This is why I proposed to carry out a more thorough evaluation of permutation methods. In particular, I contributed a non-trivial image data set that used the Signature Quadratic Form Distance (SQFD). This required writing an image feature extraction as well as an efficient implementation of the SQFD itself. I further contributed to writing the manuscript and processing the results (in particular, data related to projection quality). This evaluation was relying on our Non-Metric Space Library. My contributions in creating this library are described in section 3.2.6 concerned with Paper VI. Leonid Boytsov implemented the Neighborhood Approximation Index (NAPP) as well as the classic filter-and-refine projection method and wrote most of the text. Eric Nyberg provided support, guidance and feedback on all stages of the project (in particular, with respect to the choice of diverse realistic data sets). He also participated in writing and revising a manuscript.

3.2.2 Paper II

Bilegsaikhan Naidan and Magnus Lie Hetland, Bregman hyperplane trees for fast approximate nearest neighbor search. IJMDEM 2012.

Abstract

We present a new approximate index structure, the Bregman hyperplane tree, for indexing the Bregman divergence, aiming to decrease the number of distance computations required at query processing time, by sacrificing some accuracy in the result. The experimental results on various high-dimensional data sets demonstrate that the proposed index structure performs comparably to the state-of-the-art Bregman ball tree in terms of search performance and result quality. Moreover, our method results in a speedup of well over an order of magnitude for index construction. We also apply our space partitioning principle to the Bregman ball tree and obtain a new index structure for exact nearest neighbor search that is faster to build and a slightly slower at query processing than the original.

Research process and roles of the authors

After reading the paper for the Bregman ball tree [13], I came up with the idea to

propose hyperplane trees for Bregman divergences [10]. First I tackled the problem for exact k-NN search. However, it turned out that it is impossible. After having meetings with Øystein Torbjørnsen, Magnus Lie Hetland and Svein Erik Bratsberg, I tried only the approximate version of the tree. I did the implementation, conducted the experiments and wrote the initial version of the paper. Magnus Lie Hetland guided me how to compare approximate methods with different parameters and/or performance in a single plot and analyze the experiments. He also edited the paper. Each of the authors contributed to writing of the final versions. In the extended journal version, we applied the hyperplane partitioning to the Bregman ball tree.

3.2.3 Paper III

Bilegsaikhan Naidan and Magnus Lie Hetland, Static-to-dynamic transformation for metric indexing structures (extended version). Information Systems 2014.

Abstract

In this paper, we study the well-known algorithm of Bentley and Saxe in the context of similarity search in metric spaces. We apply the algorithm to existing static metric index structures, obtaining dynamic ones. We show that the overhead of the Bentley-Saxe method is quite low, and because it facilitates the dynamic use of any state-of-the-art static index method, we can achieve results comparable to, or even surpassing, existing dynamic methods. Another important contribution of our approach is that it is very simple—an important practical consideration. Rather than dealing with the complexities of dynamic tree structures, for example, the core index can be built statically, with full knowledge of its data set.

Research process and roles of the authors

Magnus Lie Hetland had the idea of using Bentley-Saxe method [6] for static metric indexing structures. I contributed to the main idea and came up with a parameter tuning option for deletion. I did all the implementation, conducted the experiments and wrote the initial version of the paper. Magnus Lie Hetland helped me to design the experiments, analyzed the method theoretically and edited the paper. Each of the authors discussed during the whole process and contributed to writing of the final versions. In the conference version of the paper, the experiments of deletions with dynamic spatial approximation (DSA-) tree [46] were not performed due to a bug in the SISAP metric space library [29]. However, in the journal version, we had support from one of the original authors of DSA-tree and the bug was fixed and therefore we compared our method against DSA-tree fully.

3.2.4 Paper IV

Leonid Boytsov and Bilegsaikhan Naidan, Learning to Prune in Metric and Non-Metric Spaces. NIPS 2013.

Abstract

Our focus is on approximate nearest neighbor retrieval in metric and non-metric spaces. We employ a VP-tree and explore two simple yet effective learning-to-prune approaches: density estimation through sampling and "stretching" of the triangle inequality. Both methods are evaluated using data sets with metric (Euclidean) and non-metric (KL-divergence and Itakura-Saito) distance functions. Conditions on spaces where the VP-tree is applicable are discussed. The VP-tree with a learned pruner is compared against the recently proposed state-of-the-art approaches: the bbtree, the multi-probe locality sensitive hashing (LSH), and permutation methods. Our method was competitive against state-of-the-art methods and, in most cases, was more efficient for the same rank approximation quality.

Research process and roles of the authors

This project started, because I invited Leonid to jointly work on designing and evaluating search methods for non-metric spaces. I also proposed to test our ideas using the special class of non-metric methods called Bregman divergences. For my personal research projects, I was also developing an evaluation framework that we jointly turned into what is now called the "Non-Metric Space Library". My engineering contributions are described in section 3.2.6 on Paper VI. Leonid developed a variant of the VP-tree that can tune a pruning function to the data and wrote most of the manuscript. I was helping with writing, obtaining data sets and running experiments.

3.2.5 Paper V

Bilegsaikhan Naidan and Magnus Lie Hetland, Shrinking data balls in metric indexes. DBKDA 2013.

Abstract

Some of the existing techniques for approximate similarity retrieval in metric spaces are focused on shrinking the query region by user-defined parameter. We modify this approach slightly and present a new approximation technique that shrinks data regions instead. The proposed technique can be applied to any metric indexing structure based on the ball-partitioning principle. Experiments show that our technique performs better than the relative error approximation and region proximity techniques, and that it achieves significant speedup over exact search with a low degree of error. Beyond introducing this new method, we also point

out and remedy a problem in the relative error approximation technique, substantially improving its performance.

Research process and roles of the authors

I came up with the idea, did the implementation and the experiments, and wrote the initial version of the paper. Magnus Lie Hetland helped me to design the experiments. He also added discussions about probability and distance distributions and edited the paper. Each of the authors discussed during the entire process and contributed to writing of the final versions. Øystein Torbjørnsen and Svein Erik Bratsberg offered valuable feedback during the process.

3.2.6 Paper VI

Leonid Boytsov and Bilegsaikhan Naidan, Engineering Efficient and Effective Non-metric Space Library. SISAP 2013.

Abstract

We present a new similarity search library and discuss a variety of design and performance issues related to its development. We adopt a position that engineering is equally important to design of the algorithms and pursue a goal of producing realistic benchmarks. To this end, we pay attention to various performance aspects and utilize modern hardware, which provides a high degree of parallelization. Since we focus on realistic measurements, performance of the methods should not be measured using merely the number of distance computations performed, because other costs, such as computation of a cheaper distance function, which approximates the original one, are oftentimes substantial. The paper includes preliminary experimental results, which support this point of view. Rather than looking for the best method, we want to ensure that the library implements competitive baselines, which can be useful for future work.

Research process and roles of the authors

This library (NMSLIB) is based on my personal project, which I started in 2010. It included both an evaluation framework as well as implementations of many classic metric access methods, including, but not limited to, the VP-tree, the GH-tree, list of clusters, etc. It was further extended by implementations of several permutation methods (e.g, the brute-force filtering of permutations, the MI-file, and the PP-index). Leonid proposed to make this library a full-fledged framework for searching in both metric and non-metric spaces. He also started refactoring the evaluation part and implemented efficient distance functions. At this point, I added an implementation of the BB-tree (a search method for Bregman divergences) as well a wrapper for multi-probe LSH. Leonid added an implementation of the VP-tree that can tune a pruning function to the data and, thus, be usable for a variety of non-metric spaces. He also wrote most of the manuscript: I helped

with comments and feedbacks and obtaining data sets, as well as with running experiments. I also wrote a script that generates plots for the experiments.

3.2.7 Paper VII

Bilegsaikhan Naidan and Magnus Lie Hetland, An empirical evaluation of a metric index for approximate string matching. NIK 2012.

Abstract

In this paper, we evaluate a metric index for the approximate string matching problem based on suffix trees, proposed by Gonzalo Navarro and Edgar Chávez [45]. Suffix trees are used during the index construction to generate intermediate data (pivot table) that to be indexed and the query processing. One of the main problems with suffix trees is their space requirements. To address this, we proposed as an alternative a linear-time algorithm that simulates suffix trees in the suffix arrays. The proposed algorithm is more space-efficient and is more suited for disk-based implementation. Even so, experimental results on two real-world data sets show that the metric index is beaten by straightforward, slightly enhanced linear scan.

Research process and roles of the authors

Magnus Lie Hetland introduced me the method proposed by Gonzalo Navarro and Edgar Chávez [45] and mentioned that we could use suffix arrays instead of suffix trees. I came up with a linear-time algorithm that simulates suffix trees in suffix arrays. I did all the implementation and experiments and wrote the initial version of the paper. Magnus Lie Hetaland checked the correctness of the proposed algorithm and edited the paper. Each of the authors contributed to writing of the final version.

3.3 Publication venue

Paper I was published at International Conference on Very Large Data Bases (VLDB), 2015 which is one of the top conferences in Databases [50]. So far, this paper has been cited 1 time.

The conference version of Paper II was published at International Workshop on Multimedia Databases and Data Engineering (MDDE) workshop at VLDB 2012. Eventually, it was invited to International Journal of Multimedia Data Engineering and Management (IJMDEM) after the workshop.

This conference version of Paper III was published at International Conference on Similarity Search and Applications (SISAP), 2012. SISAP is the only one con-

ference specialized in similarity search. It was selected as one of the best papers award and eventually was invited to Information Systems Journal.

Paper IV was published at Neural Information Processing Systems (NIPS), 2013 which is one of the top conferences in Artificial Intelligence [50]. This paper has been cited 2 times.

Paper V was published at International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA), 2013.

Paper VI was published at SISAP 2013 and has been cited 3 times.

Paper VII was published at Norwegian Informatics Conference (NIK), 2012.

3.4 Evaluation of contributions

This section evaluates our contributions towards the research goals which are defined in Section 1.2.

RG1. Propose new similarity search methods

- In Paper II, a new Bregman hyperplane tree for approximate nearest neighbor search under Bregman divergences was proposed. The search algorithm relies on the early termination principle which is described in Section 2.8. The performance and quality of approximate queries with our method are comparably to the state-of-the-art Bregman ball tree (BB-tree). However, our method is faster than BB-tree by an order of magnitude in index construction.
- The construction cost of the BB-tree is high. Therefore, we applied our partitioning principle to the BB-tree in Paper II.
- So far, quite a few dynamic indexing methods have been proposed. In Paper III, we presented the method of Bentley and Saxe (BS) in the context of similarity search. The BS-method allows us to transform static indexing methods into dynamic ones with less effort.
- In Paper IV, we proposed new learning to prune approaches. We applied our method to the VP-tree. Our method is shown to yield better performance than the state-of-the-art baselines in most cases.

RG2. Improve and/or evaluate the efficiency of the existing similarity search methods

- In Paper I, permutation methods were tested extensively against state-of-the-art methods on large diverse real-world metric and non-metric data sets under various distance functions, including Bregman divergences and SQFD.

- In general, indexing methods based on the ball partitioning principle (see Section 2.5) tend to have not compact balls. In Paper V, this motivated us to propose a slightly modified version of ϵ-NN technique which outperforms the relative error approximation and region proximity methods.
- In Paper VII, we evaluated a metric index for approximate string matching on DNA and protein data sets. The original method is based on suffix trees. We proposed a new algorithm that simulates suffix trees in the suffix arrays.

RG3. Broaden the applicability of similarity search methods

- As we described in Section 2.7, the VP-tree is designed only for metric spaces. In Paper IV, the VP-tree with a learned pruner is applied to non-metric Bregman divergences.
- In Paper VI, we presented so-called Non-Metric Space Library (NMSLIB) which is mainly focused on non-metric spaces, flexibility and performance.

3.5 NSMLIB in an approximate nearest neighbor benchmark

Recently, Erik Bernhardsson [8] at Spotify conducted benchmarks of approximate nearest neighbor libraries including NMSLIB, Annoy [7], FLANN [44], KGraph [23] and many popular others [24, 48, 52, 56]. At the time of writing (January 2016), the SW-graph of NMSLIB is faster than the other libraries for the same precision.

3.6 Future work

k-NN graphs are shown to be efficient. As mentioned in Section 2.11, one limitation of k-NN graph is the construction cost which is quite high for large data sets. I believe that there is a still room for improvement and it is worth to investigate how to speed up the constriction of a k-NN graph and efficient search strategies. The following ideas could be investigated:

- To my knowledge, there is no divide-and-conquer approximate k-NN construction method based on the k-means clustering algorithm. The basic idea is to apply the k-means (or any clustering) algorithm to obtain k subsets. For each subset, we build a k-NN graph using brute force method. Finally, similar to [60], all small graphs are merged into a final graph.
- Instead of the clustering based method described above, we select several pivots randomly and represent the objects as permutations. Then objects

with similar permutation prefixes end up in the same subset.

- The search algorithm for k-NN graph is often started from a random node. Instead of that, we can apply some clustering method (for instance, one in the SSS-tree) to find m objects ($m \ll n$) that are possibly far from each other. Then we call them as *fingers* which point to graph nodes from which the search can be started. For large m, we can index them using the existing metric methods. I believe that the finger-based approach would accelerate the search.

3.7 Conclusions

The main objective of this PhD thesis was to research the design and evaluation of possible strategies that can achieve efficient similarity search. We presented a number of contributions on approximate similarity search and algorithm engineering. In particular, we proposed a new index structure for Bregman divergences, a variant of ϵ-NN, and a learning-to-prune method. We surveyed state-of-the-art approximate methods. We also evaluated the Bently-Saxe method in the context of similarity search and a metric index in string matching domain. The proposed methods were evaluated on real-world data sets with different distance distributions to demonstrate their wide range of applicability.

Another practical contribution of this PhD work is the initiation of an open source research framework called the "Non Metric Space Library" (NMSLIB). We also made our data sets publicly available. I believe that our NMSLIB is useful for other researchers and commercial users as well. Therefore, I will keep contributing to the library.

Bibliography

[1] Giuseppe Amato, Fausto Rabitti, Pasquale Savino, and Pavel Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Transactions on Information Systems, TOIS*, 21(2):192–227, 2003.

[2] Giuseppe Amato and Pasquale Savino. Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd international conference on Scalable information systems*, InfoScale '08, pages 28:1–28:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[4] Sunil Arya and David M. Mount. Approximate range searching. In *Proceedings of SCG'95*, pages 172–181, 1995.

[5] Christian Beecks. *Distance based similarity models for content based multimedia retrieval*. PhD thesis, 2013.

[6] Jon Louis Bentley and James B Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301 – 358, 1980.

[7] Erik Bernhardsson. Approximate nearest neighbors in C++/Python optimized for memory usage and loading/saving to disk, 2013. `https://github.com/spotify/annoy`.

[8] Erik Bernhardsson. Benchmarks of approximate nearest neighbor libraries in Python, 2015. `https://github.com/erikbern/ann-benchmarks`.

[9] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, September 1999.

[10] L. M. Bregman. The relaxation method of finding the common points of convex sets and its application to the solution of problems in convex programming. *USSR Computational Mathematics and Mathematical Physics*, 7(3):200 – 217, 1967.

[11] Sergey Brin. Near neighbor search in large metric spaces. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of 21th International Conference on Very Large Data Bases, VLDB*, pages 574–584. Morgan Kaufmann, 1995.

[12] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *Proceedings of SOFSEM'08*, number 4910 in LNCS, pages 186–197, 2008.

[13] Lawrence Cayton. Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 112–119, 2008.

[14] Lawrence Cayton. Accelerating nearest neighbor search on manycore systems. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 402–413, May 2012.

[15] Edgar Chávez and Gonzalo Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proceedings of the 6th International Symposium on String Processing and Information Retrieval, SPIRE*, pages 75–86. IEEE Computer Society, 2000.

[16] Edgar Chávez and Gonzalo Navarro. A probabilistic spell for the curse of dimensionality. In *Revised Papers from ALENEX'01*, pages 147–160, 2001.

[17] Edgar Chávez and Gonzalo Navarro. Probabilistic proximity search: fighting the curse of dimensionality in metric spaces. *Inf. Process. Lett.*, 85(1):39–46, 2003.

[18] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[19] Lei Chen and Xiang Lian. Efficient similarity search in nonmetric spaces with local constant embedding. *Knowledge and Data Engineering, IEEE Transactions on*, 20(3):321–336, March 2008.

[20] Paolo Ciaccia and Marco Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of ICDE'00*, pages 244–255, 2000.

[21] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB*, pages 426–435. Morgan Kaufmann, 1997.

[22] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. Algorithm engineering, algorithmics column. *Bulletin of the EATCS*, 79:48–63, 2003.

[23] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM.

[24] Lyst Engineering. A forest of random projection trees, 2015. `https://github.com/lyst/rpforest`.

[25] Andrea Esuli. PP-index: Using permutation prefixes for efficient and scalable approximate similarity search. In *Proceedings of LSDS-IR*, 2009.

[26] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A content–addressable network for similarity search in metric spaces. In *Databases, Information Systems, and Peer-to-Peer Computing: International Workshops, DBISP2P 2005/2006*, pages 98–110, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[27] Christos Faloutsos and King-Ip Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia. In *Proceedings of SIGMOD'95*, pages 163–174, 1995.

[28] Karina Figueroa and Kimmo Fredriksson. Speeding up permutation based indexing with indexing. In *Similarity Search and Applications, 2009. SISAP '09. Second International Workshop on*, pages 107–114, Aug 2009.

[29] Karina Figueroa, Gonzalo Navarro, and Edgar Chavez. Metric spaces library, 2010. `http://www.sisap.org/Metric_Space_Library.html`.

[30] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[31] K. Fukunage and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, 24(7):750–753, July 1975.

[32] Edgar Chavez Gonzalez, Karina Figueroa, and Gonzalo Navarro. Effective proximity retrieval by ordering permutations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(9):1647–1658, 2008.

[33] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, pages 47–57, 1984.

[34] Magnus Lie Hetland. The basic principles of metric indexing. In C. A. Coello Coello, S. Dehuri, and S. Ghosh, editors, *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*. Springer, 2009.

[35] Magnus Lie Hetland. Ptolemaic indexing. *Journal of Computational Geometry*, 2015.

[36] Magnus Lie Hetland, Tomáš Skopal, Jakub Lokoč, and Christian Beecks. Ptolemaic access methods: Challenging the reign of the metric space model. *Information Systems*, 38(7):989 – 1006, 2013.

[37] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems, TODS*, 28(4):517–580, December 2003.

[38] Piotr Indyk. Nearest neighbors in high-dimensional spaces. In Jacob E Goodman and Joseph O'Rourke, editors, *Handbook of discrete and computational geometry*, pages 877–892. Chapman and Hall/CRC, 2004.

[39] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[40] David W. Jacobs, Daphna Weinshall, and Yoram Gdalyahu. Classification with nonmetric distances: Image retrieval and class representation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(6):583–600, June 2000.

[41] Ting Liu, Andrew W. Moore, Er Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[42] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61 – 68, 2014.

[43] Bernard M. E. Moret. Towards a discipline of experimental algorithmics.

[44] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[45] Gonzalo Navarro and Edgar Chávez. A metric index for approximate string matching. *Theoretical Computer Science*, 352:266–279, March 2006.

[46] Gonzalo Navarro and Nora Reyes. Dynamic spatial approximation trees. *Journal of Experimental Algorithmics (JEA)*, 12:1.5:1–1.5:68, 2008.

[47] David Novak and Pavel Zezula. M-chord: a scalable distributed similarity search structure. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, 2006.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[49] Volker Roth, Julian Laub, Joachim M. Buhmann, and Klaus R. Müller. Going metric: Denoising pairwise data. In *Proceedings of NIPS*, 2002.

[50] Microsoft Academic Search. Top conferences in computer science. `http://academic.research.microsoft.com/RankList?entitytype=3&topDomainID=2`, Last checked on November 26th, 2015.

[51] Tomáš Skopal. Unified framework for exact and approximate search in dissimilarity spaces. *ACM Transactions on Database Systems, TODS*, 32(4), 2007.

[52] Ole Sprause. Ann search in large, high-dimensional data sets (in python), 2013. `http://nearpy.io`.

[53] Eric Sadit Tellez, Edgar Chavez, and Gonzalo Navarro. Succinct nearest neighbor search. *Information Systems*, 38(7):1019–1030, 2013.

[54] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[55] J. T. L. Wang, Xiong Wang, D. Shasha, and Kaizhong Zhang. Metricmap: an embedding technique for processing distance-based queries in metric spaces. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 35(5):973–987, October 2005.

[56] Liang Wang. Python approximate nearest neighbour search in high-dimension space with optimised indexing, 2014. `http://www.cl.cam.ac.uk/~lw525/`.

[57] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[58] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search : The Metric Space Approach*. Springer, 2006.

[59] Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. Approximate similarity retrieval with M-Trees. *The VLDB Journal*, 7(4):275–293, 1998.

[60] Yan-Ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-Lin Liu. Fast knn graph construction with locality sensitive hashing. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 8189 of *Lecture Notes in Computer Science*, pages 660–674. Springer Berlin Heidelberg, 2013.

Part II

Publications

Appendix A

Paper I: Permutation Search Methods are Efficient, Yet Faster Search is Possible

Bilegsaikhan Naidan, Leonid Boytsov and Eric Nyberg.
Appeared at the Proceedings of the VLDB Endowment, 2015.

Permutation Search Methods are Efficient, Yet Faster Search is Possible

Bilegsaikhan Naidan*
Norwegian University of Science and Technology
Trondheim, Norway
bileg@idi.ntnu.no

Leonid Boytsov
Carnegie Mellon University
Pittsburgh, PA, USA
srchvrs@cs.cmu.edu

Eric Nyberg
Carnegie Mellon University
Pittsburgh, PA, USA
ehn@cs.cmu.edu

ABSTRACT

We survey permutation-based methods for approximate k-nearest neighbor search. In these methods, every data point is represented by a ranked list of pivots sorted by the distance to this point. Such ranked lists are called *permutations*. The underpinning assumption is that, for both metric and non-metric spaces, the distance between permutations is a good proxy for the distance between original points. Thus, it should be possible to efficiently retrieve most true nearest neighbors by examining only a tiny subset of data points whose permutations are similar to the permutation of a query. We further test this assumption by carrying out an extensive experimental evaluation where permutation methods are pitted against state-of-the art benchmarks (the multi-probe LSH, the VP-tree, and proximity-graph based retrieval) on a variety of realistically large data set from the image and textual domain. The focus is on the high-accuracy retrieval methods for generic spaces. Additionally, we assume that both data and indices are stored in main memory. We find permutation methods to be reasonably efficient and describe a setup where these methods are most useful. To ease reproducibility, we make our software and data sets publicly available.

1. INTRODUCTION

Nearest-neighbor searching is a fundamental operation employed in many applied areas including pattern recognition, computer vision, multimedia retrieval, computational biology, and statistical machine learning. To automate the search task, real-world objects are represented in a compact numerical, e.g., vectorial, form and a distance function $d(x, y)$, e.g., the Euclidean metric L_2, is used to evaluate the similarity of data points x and y. Traditionally, it assumed that the distance function is a non-negative function that is small for similar objects and large for dissimilar one. It is equal to zero for identical x and y and is always positive when objects are different.

This mathematical formulation allows us to define the nearest-neighbor search as a *conceptually simple* optimization procedure. Specifically, given a query data point q, the goal is to identify the nearest (neighbor) data point x, i.e., the point with the minimum distance value $d(x, q)$ among all data points (ties can be resolved arbitrarily). A natural generalization is a k-NN search, where we aim to find k closest points instead of merely one. If the distance is not symmetric, two types of queries are considered: *left* and *right* queries. In a *left* query, a data point compared to the query is always the first (i.e., the left) argument of $d(x, y)$.

Despite being conceptually simple, finding nearest neighbors in efficient and effective fashion is a *notoriously hard* task, which has been a recurrent topic in the database community (see e.g. [43, 20, 2, 28]). The most studied instance of the problem is an *exact* nearest-neighbor search in vector spaces, where a distance function is an actual metric distance (a non-negative, symmetric function satisfying the triangle inequality). If the search is exact, we must guarantee that an algorithm *always* finds a true nearest-neighbor no matter how much computational resources such a quest may require. Comprehensive reviews of exact approaches for metric and/or vector spaces can be found in books by Zezula et al. [45] and Samet [35].

Yet, exact methods work well only in low dimensional metric spaces.[1] Experiments showed that exact methods can rarely outperform the sequential scan when dimensionality exceeds ten [43]. This a well-known phenomenon known as "the curse of dimensionality".

Furthermore, a lot of applications are increasingly relying on non-metric spaces (for a list of references related to computer vision see, e.g., a work by Jacobs et al. [25]). This is primarily because many problems are inherently non-metric [25]. Thus, using, a non-metric distance permits sometimes a better representation for a domain of interest. Unfortunately, exact methods for metric-spaces are not directly applicable to non-metric domains.

Compared to metric spaces, it is more difficult to design exact methods for arbitrary non-metric spaces, in particular, because they lack sufficiently generic yet simple properties such as the triangle inequality. When exact search methods for non-metric spaces do exist, they also seem to suffer from the curse of dimensionality [10, 9].

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

[1] A dimensionality of a vector space is simply a number of coordinates necessary to represent a vector: This notion can be generalized to metric spaces without coordinates [12].

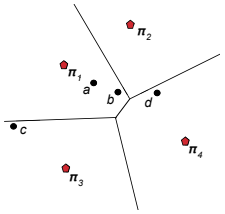


Figure 1: Voronoi diagram produced by four pivots π_i. The data points are a, b, c, and d. The distance is L_2.

Approximate search methods are less affected by the curse of dimensionality [31] and can be used in various non-metric spaces when exact retrieval is not necessary [38, 23, 13, 10, 9]. Approximate search methods can be much more efficient than exact ones, but this comes at the expense of a reduced search accuracy. The quality of approximate searching is often measured using *recall*, which is equal to the average fraction of true neighbors returned by a search method. For example, if the method routinely misses every other true neighbor, the respective recall value is 50%.

Permutation-based algorithms is an important class of approximate retrieval methods that was independently introduced by Amato [3] and Chávez et al. [24]. It is based on the idea that if we rank a set of reference points–called *pivots*–with respect to distances from a given point, the pivot rankings produced by two near points should be similar. A number of methods based on this idea were recently proposed and evaluated [3, 24, 19, 11, 2] (these methods are briefly surveyed in § 2). However, a comprehensive evaluation that involves a diverse set of large metric and non-metric data sets (i.e., asymmetric and/or hard-to-compute distances) is lacking. In § 3, we fill this gap by carrying out an extensive experimental evaluation where these methods (implemented by us) are compared against some of the most efficient state-of-the art benchmarks. The focus is on the high-accuracy retrieval methods (recall close to 0.9) for generic spaces. Because distributed high-throughput main memory databases are gaining popularity (see., e.g. [27]), we focus on the case where data and indices are stored in main memory. Potentially, the data set can be huge, yet, we run experiments only with a smaller subset that fits into a memory of one server.

2. PERMUTATION METHODS

2.1 Core Principles

Permutation methods are *filter-and-refine* methods belonging to the class of pivoting searching techniques. *Pivots* (henceforth denoted as π_i) are reference points randomly selected during indexing. To create an index, we compute the distance from every data point x to every pivot π_i. We then memorize either the original distances or some distance statistics in the hope that these statistics can be useful during searching. At search time, we compute distances from the query to pivots and prune data points using, e.g., the triangle inequality [35, 45] or its generalization for non-metric spaces [21].

Alternatively, rather than relying on distance values directly, we can use precomputed statistics to produce estimates for distances between the query and data points. In particular, in the case of permutation methods, we assess similarity of objects based on their relative distances to pivots. To this end, for each data point x, we arrange pivots π_i in the order of increasing distances from x. The ties can be resolved, e.g., by selecting a pivot with the smallest index. Such a *permutation* (i.e., ranking) of pivots is essentially a vector whose i-th element keeps an ordinal position of the i-th pivot in the set of pivots sorted by their distances from x. We say that point x *induces* the permutation.

Consider the Voronoi diagram in Figure 1 produced by pivots π_1, π_2, π_3, and π_4. Each pivot π_i is associated with its own cell containing points that are *closer* to π_i than to any other pivot $\pi_j, i \neq j$. The neighboring cells of two pivots are separated by a segment of the line *equidistant* to these pivots. Each of the data points a, b, c, and d "sits" in the cell of its closest pivot.

For the data point a, points π_1, π_2, π_3, and π_4 are respectively the first, the third, and the forth closest pivots. Therefore, the point a induces the permutation $(1, 2, 3, 4)$. For the data point b, which is the nearest neighbor of a, two closest pivots are also π_1 and π_2. However, π_4 is closer than π_3. Therefore, the permutation induced by b is $(1, 2, 4, 3)$. Likewise, the permutations induced by c and d are $(2, 3, 1, 4)$ and $(3, 2, 4, 1)$, respectively.

The *underpinning assumption* of permutation methods is that most nearest neighbors can be found by retrieving a small fraction of data points whose pivot rankings, i.e., the induced permutations, are similar to the pivot ranking of the query. Two most popular choices to compare the rankings x and y are: Spearman's rho distance (equal to the squared L_2) and the Footrule distance (equal to L_1) [14, 24]. More formally, $\text{SpearmanRho}(x, y) = \sum_i (x_i - y_i)^2$ and $\text{Footrule}(x, y) = \sum_i |x_i - y_i|$. According to Chávez et al. [24] Spearman's rho is more effective than the Footrule distance. This was also confirmed by our own experiments.

Converting the vector of distances to pivots into a permutation entails information loss, but this loss is not necessarily detrimental. In particular, our preliminary experiments showed that using permutations instead of vectors of original distances results in slightly better retrieval performance. The information about relative positions of the pivots can be further coarsened by binarization: All elements smaller than a threshold b become zeros and elements at least as large as b become ones [39]. The similarity of binarized permutations is computed via the Hamming distance.

In the example of Figure 1, the values of the Footrule distance between the permutation of a and permutations of b, c, and d are equal to 2, 4, and 6, respectively. Note that the Footrule distance on permutations correctly "predicts" the closest neighbor of a. Yet, the ordering of points based on the Footrule distance is not perfect: the Footrule distance from the permutation of a to the permutation of its second nearest neighbor d is larger than the Footrule distance to the permutation of the third nearest neighbor c.

Given the threshold $b = 3$, the binarized permutations induced by a, b, c, and d are equal to $(0, 0, 1, 1)$, $(0, 0, 1, 1)$, $(0, 1, 0, 1)$, and $(1, 0, 1, 0)$, respectively. In this example, the binarized permutation of a and its nearest neighbor b are equal, i.e., the distance between respective permutations is zero. When we compare a to c and d, the Hamming distance does not discriminate between c and d as their binary

permutations are both at distance two from the binary permutation of a.

Permutation-based searching belongs to a class of *filter-and-refine* methods, where objects are mapped to data points in a low-dimensional space (usually L_1 or L_2). Given a permutation of a query, we carry out a nearest neighbor search in the space of permutations. Retrieved entries represent a (hopefully) small list of candidate data points that are compared directly to the query using the distance in the *original* space. The permutation methods differ in ways of producing candidate records, i.e., in the way of carrying out the filtering step. In the next sections we describe these methods in detail.

Permutation methods are similar to the rank-aggregation method OMEDRANK due to Fagin et al. [20]. In OMEDRANK there is a small set of voting pivots, each of which ranks data points based on a somewhat imperfect notion of the distance from points to the query (e.g., computed via a random projection). While each individual ranking is imperfect, a more accurate ranking can be achieved by rank aggregation. Thus, unlike permutation methods, OMEDRANK uses pivots to rank data points and aims to find an *unknown* permutation of *data points* that reconciles differences in data point rankings in the best possible way. When such a consolidating ranking is found, the most highly ranked objects from this *aggregate* ranking can be used as answers to a nearest-neighbor query. Finding the aggregate ranking is an NP-complete problem that Fagin et al. [20] solve only heuristically. In contrast, permutation methods use data points to rank pivots and solve a much simpler problem of finding *already known and computed* permutations of *pivots* that are the best matches for the query permutation.

2.2 Brute-force Searching of Permutations

In this approach, the filtering stage is implemented as a brute-force comparison of the query permutation against the permutations of the data with subsequent selection of the γ entries that are γ-nearest objects in the space of permutations. A number of candidate entries γ is a parameter of the search algorithm that is often understood as a fraction (or percentage) of the total number of points. Because the distance in the space of permutations is not a perfect proxy for the original distance, to answer a k-NN-query with high accuracy, the number of candidate records has to be much larger than k (see § 3.4).

A straightforward implementation of brute-force searching relies on a priority queue. Chávez et al. [24] proposed to use incremental sorting as a more efficient alternative. In our experiments with the L_2 distance, the latter approach is twice as fast as the approach relying on a standard C++ implementation of a priority queue.

The cost of the filtering stage can be reduced by using binarized permutations [39]. Binarized permutations can be stored compactly as bit arrays. Computing the Hamming distance between bit arrays can be done efficiently by XORing corresponding computer words and counting the number of non-zero bits of the result. For bit-counting, one can use a special instruction available on many modern CPUs. [3]

The brute-force searching in the permutation space, unfortunately, is not very efficient, especially if the distance can be easily computed: If the distance is "cheap" (e.g., L_2) and the index is stored in main memory, the brute-force search in the space of permutations is not much faster than the brute-force search in the original space.

[3]In C++, this instruction is provided via the intrinsic function `__builtin_popcount`.

2.3 Indexing of Permutations

To reduce the cost of the filtering stage of permutation-based searching, three types of indices were proposed: the Permutation Prefix Index (PP-Index) [19], existing methods for metric spaces [22], and the Metric Inverted File (MI-file) [3].

Permutations are integer vectors whose values are between one and the total number of pivots m. We can view these vectors as sequences of symbols over a finite alphabet and index these sequences using a prefix tree. This approach is implemented in the PP-index. At query time, the method aims to retrieve γ candidates by finding permutations that share a prefix of a given length with the permutation of the query object. This operation can be carried out efficiently via the prefix tree constructed at index time. If the search generates fewer candidates than a specified threshold γ, the procedure is recursively repeated using a shorter prefix. For example, the permutations of points a, b, c, and d in Figure 1 can be seen as strings `1234`, `1243`, `2314`, and `3241`. The permutation of points a and b, which are nearest neighbors, share a two-character prefix with a. In contrast, permutations of points c and d, which are more distant from a than b, have no common prefix with a.

To achieve good recall, it may be necessary to use short prefixes. However, longer prefixes are more selective than shorter ones (i.e., they generate fewer candidate records) and are, therefore, preferred for efficiency reasons. In practice, a good trade-off between recall and efficiency is typically achieved only by building several copies of the PP-index (using different subsets of pivots) [2].

Figueroa and Fredriksson experimented with indexing permutations using well-known data structures for metric spaces [22]. Indeed, the most commonly used permutation distance: Spearman's rho, is a monotonic transformation (squaring) of the Euclidean distance. Thus, it should be possible to find γ nearest neighbors by indexing permutations, e.g., in a VP-tree [44, 41].

Amato and Savino proposed to index permutation using an inverted file [3]. They called their method the MI-file. To build the MI-file, they first select m pivots and compute their permutations/rankings induced by data points. For each data point, $m_i \leq m$ most closest pivots are indexed in the inverted file. Each posting is a pair $(pos(\pi_i, x), x)$, where x is the identifier of the data point and $pos(\pi_i, x)$ is a position of the pivot in the permutation induced by x. Postings of the same pivot are sorted by pivot's positions.

Consider the example of Figure 1 and imagine that we index two most closest pivots (i.e., $m_i = 2$). The point a induces the permutation $(1, 2, 3, 4)$. Two closest pivots π_1 and π_2 generate postings $(1, a)$ and $(2, a)$. The point b induces the permutation $(1, 2, 4, 3)$. Again, π_1 and π_2 are two pivots closest to b. The respective postings are $(1, b)$ and $(2, b)$. The permutation of c is $(2, 3, 1, 4)$. Two closest pivots are π_1 and π_3. The respective postings are $(2, c)$ and $(1, c)$. The permutation of d is $(3, 2, 4, 1)$. Two closest pivots are π_2 and π_4 with corresponding postings $(2, d)$ and $(1, d)$.

At query time, we select $m_s \leq m_i$ pivots closest to the query q and retrieve respective posting lists. If $m_s = m_i = m$, it is possible to compute the exact Footrule distance (or

Table 1: Summary of Data Sets

Name	Distance function	# of rec.	Brute-force search (sec)	In-memory size	Dimens.	Source
Metric Data						
CoPhIR	L_2	$5 \cdot 10^6$	0.6	5.4GB	282	MPEG7 descriptors [7]
SIFT	L_2	$5 \cdot 10^6$	0.3	2.4GB	128	SIFT descriptors [26]
ImageNet	SQFD[4]	$1 \cdot 10^6$	4.1	0.6 GB	N/A	Signatures generated from ImageNet LSVRC-2014 [34]
Non-Metric Data						
Wiki-sparse	Cosine sim.	$4 \cdot 10^6$	1.9	3.8GB	10^5	Wikipedia TF-IDF vectors generated via Gensim [33]
Wiki-8	KL-div/JS-div	$2 \cdot 10^6$	0.045/0.28	0.13GB	8	LDA (8 topics) generated from Wikipedia via Gensim [33]
Wiki-128	KL-div/JS-div	$2 \cdot 10^6$	0.22/4	2.1GB	128	LDA (128 topics) generated from Wikipedia via Gensim [33]
DNA	Normalized Levenshtein	$1 \cdot 10^6$	3.5	0.03GB	N/A	Sampled from the Human Genome[2] with sequence length $\mathcal{N}(32, 4)$

Spearman's rho) between the query permutation and the permutation induced by data points. One possible search algorithm keeps an accumulator (initially set to zero) for every data point. Posting lists are read one by one: For every encountered posting $(pos(\pi_i, x), x)$ we increase the accumulator of x by the value $|pos(\pi_i, x) - pos(\pi_i, q)|$. If the goal is to compute Spearman's rho, the accumulator is increased by $|pos(\pi_i, x) - pos(\pi_i, q)|^2$.

If $m_s < m$, by construction of the posting lists, using the inverted index, it is possible to obtain rankings of only $m_s < m$ pivots. For the remaining, $m - m_s$ pivots we pessimistically assume that their rankings are all equal to m (the maximum possible value). Unlike the case $m_i = m_s = m$, all accumulators are initially set to $m_s \cdot m$. Whenever we encounter a posting posting $(pos(\pi_i, x), x)$ we subtract $m - |pos(\pi_i, x) - pos(\pi_i, q)|$ from the accumulator of x.

Consider again the example of Figure 1. Let $m_i = m_s = 2$ and a be the query point. Initially, the accumulators of b, c, and d contain values $4 \cdot 2 = 8$. Because $m_s = 2$, we read posting lists only of the two closest pivots for the query point a, i.e., π_1 and π_2. The posting lists of π_1 is comprised of $(1, a)$, $(1, b)$, and $(2, c)$. On reading them (and ignoring postings related to the query a), accumulators b and c are decreased by $4 - |1 - 1| = 4$ and $4 - |1 - 2| = 3$, respectively. The posting lists of π_2 are $(2, a)$, $(2, b)$, and $(2, d)$. On reading them, we subtract $4 - |2 - 2| = 4$ from each of the accumulators b and d. In the end, the accumulators b, c, d are equal to 0, 5, and 4. Unlike the case when we compute the Footrule distance between complete permutation, the Footrule distance on truncated permutations correctly predicts the order of three nearest neighbors of a.

Using fewer pivots at retrieval time allows us to reduce the number of processed posting lists. Another optimization consists in keeping posting lists sorted by pivots position $pos(\pi_i, x)$ and retrieving only the entries satisfying the following restriction on the maximum position difference: $|pos(\pi_i, x) - pos(\pi_i, q)| \leq D$, where D is a method parameter. Because posting list entries are sorted by pivot positions, the first and the last entry satisfying the maximum position difference requirement can be efficiently found via the binary search.

Tellez et al. [40] proposed a modification of the MI-file which they called a Neighborhood APProximation index (NAPP). In the case of NAPP, there also exist a large set of m pivots of which only $m_i < m$ pivots (most closest to inducing data points) are indexed. Unlike the MI-file, however, posting lists contain only object identifiers, but no positions of pivots in permutations. Thus, it is not possible to compute an estimate for the Footrule distance by reading only posting lists. Therefore, instead of an estimate for the Footrule distance, the number of most closest *common* pivots is used to sort candidate objects. In addition, the candidate objects sharing with the query fewer than t closest pivots are discarded (t is a parameter). For example, points a and b in Figure 1 share the same common pivot π_1. At the same time a does not share any closest pivot with points d and c. Therefore, if we use a as a query, the point b will be considered to be the best candidate point.

Chávez et al. [24] proposed a single framework that unifies several approaches including PP-index, MI-file, and NAPP. Similar to the PP-index, permutations are viewed as strings over a finite alphabet. However, these strings are indexed using a special sequence index (rather than a prefix tree) that efficiently supports rank and select operations. These operations can be used to simulate various index traversal modes, including, e.g., retrieval of all strings whose i-th symbol is equal to a given one.

3. EXPERIMENTS

3.1 Data Sets and Distance Functions

We employ three image data sets: CoPhIR, SIFT, ImageNet, and several data sets created from textual data. The smallest data set (DNA) has one million entries, while the largest one (CoPhIR) contains five million high-dimensional vectors. All data sets derived from Wikipedia were generated using the topic modelling library GENSIM [33]. The data set meta data is summarized in Table 1. Below, we describe our data sets in detail.

CoPhIR is a five million subset of MPEG7 descriptors downloaded from the website of the Institute of the National Research Council of Italy[7].

SIFT is a five million subset of SIFT descriptors (from the learning subset) downloaded from a TEXMEX collection website[26].[4]

In experiments involving CoPhIR and SIFT, we employed L_2 to compare unmodified, i.e., raw visual descriptors. We implemented an optimized procedure to compute L_2 that relies on Single Instruction Multiple Data (SIMD) operations available on Intel-compatible CPUs. Using this implementation, it is possible to carry out about 20 million L_2 computations per second using SIFT vectors or 10 million L_2 computations using CoPhIR vectors.

ImageNet collection comprises one million signatures extracted from LSVRC-2014 data set [34], which contains 1.2 million high resolution images. We implemented our own code to extract signatures following the method of Beecks [4]. For each image, we selected 10^4 pixels randomly and mapped them into 7-dimensional feature space: three color, two position, and two texture dimensions.

The features were clustered by the standard k-means algorithm with 20 clusters. Then, each cluster was represented by an 8-dimensional vector, which included a 7-dimensional centroid and a cluster weight (the number of cluster points divided by 10^4).

Images were compared using a metric function called the Signature Quadratic Form Distance (SQFD). This distance is computed as a quadratic form, where the matrix is recomputed for each pair of images using a heuristic similarity function applied to cluster representatives. It is a distance metric defined over vectors from an infinite-dimensional space such that each vector has only finite number of non-zero elements. For further details, please, see the thesis of Beecks [4]. SQFD was shown to be effective [4]. Yet, it is nearly two orders of magnitude slower compared to L_2.

Wiki-sparse is a set of four million sparse TF-IDF vectors (created via GENSIM [33]). On average, these vectors have 150 non-zero elements out of 10^5. Here we use a cosine similarity, which is a symmetric non-metric distance:

$$d(x,y) = 1 - \left(\sum_{i=1}^{n} x_i y_i\right)\left(\sum_{i=1}^{n} x_i^2\right)^{-1/2}\left(\sum_{i=1}^{n} y_i^2\right)^{-1/2}.$$

Computation of the cosine similarity between sparse vectors relies on an efficient procedure to obtain an intersection of non-zero element indices. To this end, we use an all-against-all SIMD comparison instruction as was suggested by Schlegel et al. [36]. This distance function is relatively fast being only about 5x slower compared to L_2.

Wiki-i consist of dense vectors of topic histograms created using the Latent Dirichlet Allocation (LDA)[6]. The index $i \in \{8, 128\}$ denotes the number of topics. To create these sets, we trained a model on one half of the Wikipedia collection and then applied it to the other half (again using GENSIM [33]). Zero values were replaced by small numbers (10^{-5}) to avoid division by zero in the distance calculations. Two distance functions were used for these data sets: the Kullback-Leibler (KL) divergence: $\sum_{i=1}^{n} x_i \log \frac{x_i}{y_i}$ and its symmetrized version called the Jensen-Shannon (JS) divergence:

$$d(x,y) = \frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i)\log\frac{x_i + y_i}{2}\right].$$

Both the KL- and the JS-divergence are non-metric distances. Note that the KL-divergence is not even symmetric.

Our implementation of the KL-divergence relies on the precomputation of logarithms at index time. Therefore, during retrieval it is as fast as L_2. In the case of JS-divergence, it is not possible to precompute $\log(x_i + y_i)$ and, thus, it is about 10-20 times slower compared to L_2.

DNA is a collection of DNA sequences sampled from the Human Genome [5]. Starting locations were selected randomly and uniformly (however, lines containing the symbol `N` were excluded). The length of the sequence was sampled from $\mathcal{N}(32, 4)$. The employed distance function was the *normalized Levenshtein distance*. This non-metric distance is equal to the minimum number of edit operations (insertions, deletions, substitutions), needed to convert one sequence into another, divided by the maximum of the sequence lengths.

3.2 Tested Methods

Table 2 lists all implemented methods and provides information on index creation time and size.

Multiprobe-LSH (MPLSH) is implemented in the library LSHKit [6]. It is designed to work only for L_2. Some parameters are selected automatically using the cost model proposed by Dong et al. [17]. However, the size of the hash table H, the number of hash tables L, and the number of probes T need to be chosen manually. We previously found that (1) $L = 50$ and $T = 10$ provided a near optimal performance and (2) performance is not affected much by small changes in L and T [9]. This time, we re-confirmed this observation by running a small-scale grid search in the vicinity of $L = 50$ and $T = 50$ for H equal to the number of points plus one. The MPLSH generates a list of candidates that are directly compared against the query. This comparison involves the optimized SIMD implementation of L_2.

VP-tree is a classic space decomposition tree that recursively divides the space with respect to a randomly chosen pivot π[44, 41]. For each partition, we compute a median value R of the distance from π to every other point in the current partition. The pivot-centered ball with the radius R is used to partition the space: the inner points are placed into the left subtree, while the outer points are placed into the right subtree (points that are exactly at distance R from π can be placed arbitrarily).

Partitioning stops when the number of points falls below the threshold b. The remaining points are organized in a form of a bucket. In our implementation, all points in a bucket are stored in the same chunk of memory. For cheap distances (e.g., L_2 and KL-div) this placing strategy can halve retrieval time.

If the distance is the metric, the triangle inequality can be used to prune unpromising partitions as follows: imagine that r is a radius of the query and the query point is inside the pivot-centered ball (i.e., in the left subtree). If $R - d(\pi, q) > r$, the right partition cannot have an answer, i.e., the right subtree can be safely pruned. If the query point is in the right partition, we can prune the left subtree if $d(\pi, q) - R > r$. The nearest-neighbor search is simulated as a range search with a decreasing radius: Each time we evaluate the distance between q and a data point, we

[4] `http://corpus-texmex.irisa.fr/`

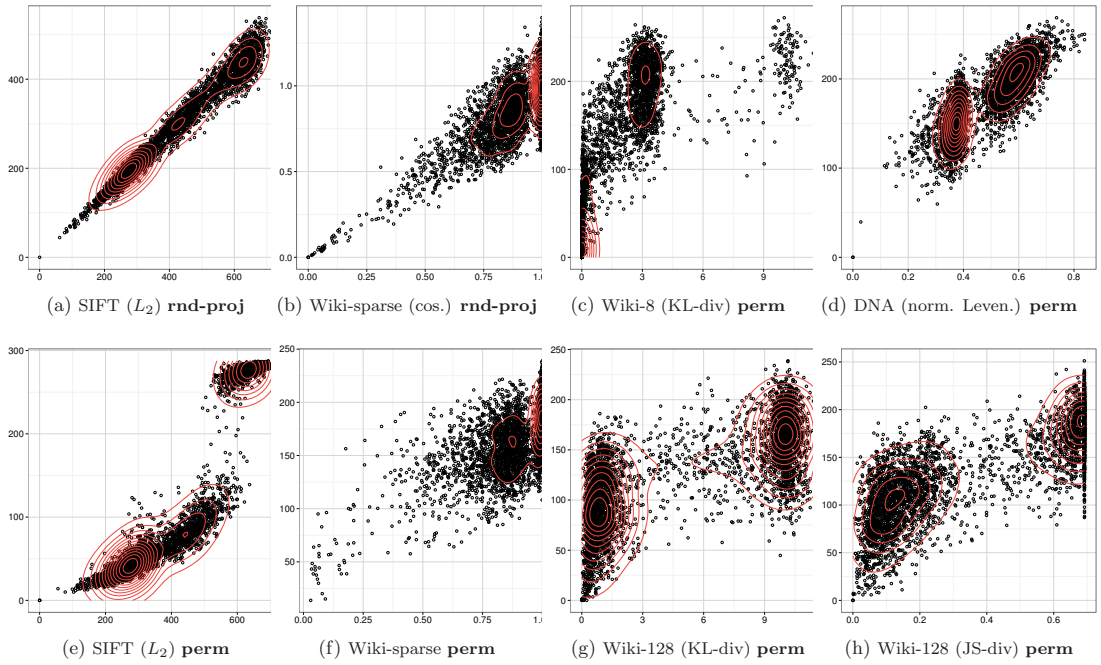
[5] `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/`

[6] Downloaded from `http://lshkit.sourceforge.net/`

Table 2: Index Size and Creation Time for Various Data Sets

	VP-tree	NAPP	LSH	Brute-force filt.	k-NN graph
Metric Data					
CoPhIR	5.4 GB (0.5min)	6 GB (6.8min)	13.5 GB (23.4min)		7 GB (52.1min)
SIFT	2.4 GB (0.4min)	3.1 GB (5min)	10.6 GB (18.4min)		4.4 GB (52.2min)
ImageNet	1.2 GB (4.4min)	0.91 GB (33min)		12.2 GB (32.3min)	1.1 GB (127.6min)
Non-Metric Data					
Wiki-sparse		4.4 GB (7.9min)			5.9 GB (231.2min)
Wiki-8 (KL-div)	0.35 GB (0.1min)	0.67 GB (1.7min)			962 MB (11.3min)
Wiki-128 (KL-div)	2.1 GB (0.2min)	2.5 GB (3.1min)			2.9 GB (14.3min)
Wiki-8 (JS-div)	0.35 GB (0.1min)	0.67 GB (3.6min)			2.4 GB (89min)
Wiki-128 (JS-div)	2.1 GB (1.2min)	2.5 GB (36.6min)			2.8 GB (36.1min)
DNA	0.13 GB (0.9min)	0.32 GB (15.9min)		61 MB (15.6min)	1.1 GB (88min)

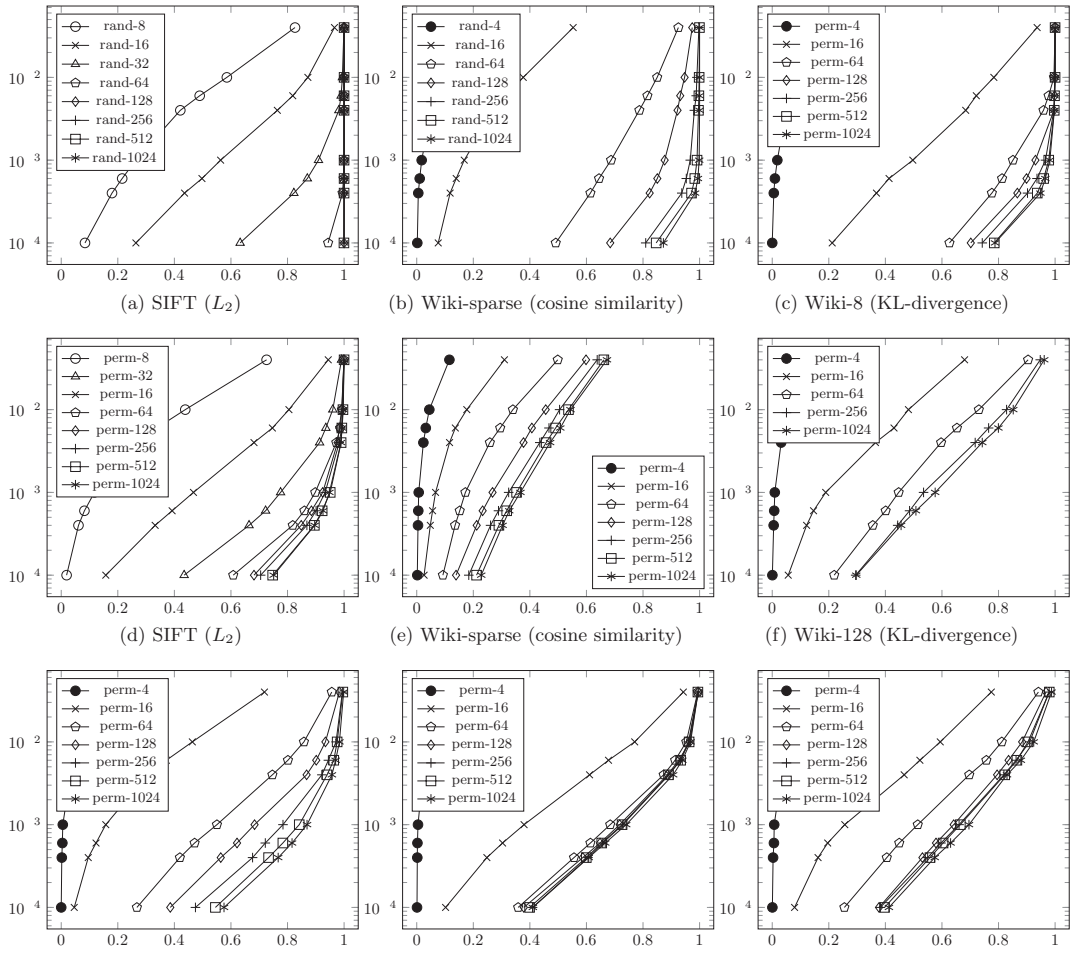
Note: The indexing algorithms of NAPP and k-NN graphs used four threads.
In all but two cases (DNA and Wiki-8 with JS-divergence), we build the k-NN graph using the Small World algorithm [29].
In the case of DNA or Wiki-8 with JS-divergence, we build the k-NN graph using the NN-descent algorithm [16].



(a) SIFT (L_2) **rnd-proj** (b) Wiki-sparse (cos.) **rnd-proj** (c) Wiki-8 (KL-div) **perm** (d) DNA (norm. Leven.) **perm**

(e) SIFT (L_2) **perm** (f) Wiki-sparse **perm** (g) Wiki-128 (KL-div) **perm** (h) Wiki-128 (JS-div) **perm**

Figure 2: Distance values in the projected space (on the y-axis) plotted against original distance values (on the x-axis). Plots 2a and 2b use random projections. The remaining plots rely on permutations. Dimensionality of the target space is 64. All plots except Plot 2b represent projections to L_2. In Plot 2b the target distance function is the cosine similarity. Distances are computed for pairs of points sampled at random. Sampling is conducted from two strata: a complete subset and a set of points that are 100-nearest neighbors of randomly selected points. All data sets have one million entries.



(a) SIFT (L_2) (b) Wiki-sparse (cosine similarity) (c) Wiki-8 (KL-divergence)

(d) SIFT (L_2) (e) Wiki-sparse (cosine similarity) (f) Wiki-128 (KL-divergence)

(g) DNA (Normalized Levenshtein) (h) ImageNet (SQFD) (i) Wiki-128 (JS-divergence)

Figure 3: A fraction of candidate records that are necessary to retrieve to ensure a desired recall level (10-NN search). The candidate entries are ranked in a projected space using either the cosine similarity (only for Wiki-sparse) or L_2 (for all the other data sets). Two types of projections are used: random projections (rand) and permutations (perm). In each plot, there are several lines that represent projections of different dimensionality. Each data (sub)set in this experiment contains one million entries.

compare this distance with r. If the distance is smaller, it becomes a new value of r. In the course of traversal, we first visit the closest subspace (e.g., the left subtree if the query is inside the pivot-centered ball).

For a generic, i.e., not necessarily metric, space, the pruning conditions can be modified. For example, previously we used a liner "stretching" of the triangle inequality [9]. In this work, we employed a simple polynomial pruner. More specifically, the right partition can be pruned if the query is in the left partition and $(R - d(\pi, q))^{\beta}\alpha_{left} > r$. The left partition can be pruned if the query is in the right partition and $(d(\pi, q) - R)^{\beta}\alpha_{right} > r$.

We used $\beta = 2$ for the KL-divergence and $\beta = 1$ for every other distance function. The optimal parameters α_{left} and α_{right} can be found by a trivial grid-search-like procedure with a shrinking grid step [9] (using a subset of data).

k-NN graph (a proximity graph) is a data structure in which data points are associated with graph nodes and k edges are connected to k nearest neighbors of the node. The search algorithm relies on a concept "the closest neighbor of my closest neighbor is my neighbor as well." This algorithm can start at an arbitrary node and recursively transition to a neighbor point (by following the graph edge) that is closest to the query. This greedy algorithm stops when the current point x is closer to the query than any of the x's neighbors. However, this algorithm can be trapped in a local minima [15]. Alternatively, the termination condition can be defined in terms of an extended neighborhood [37, 29].

Constructing an *exact* k-NN graph is hardly feasible for a large data set, because, in the worst case, the number of distance computations is $O(n^2)$, where n in the number of data points. While there are amenable metric spaces where an exact graph can be computed more efficiently than in $O(n^2)$, see e.g. [30], the quadratic cost appear to be unavoidable in many cases, especially if the distance is not a metric or the intrinsic dimensionality is high.

An *approximate* k-NN graph can be constructed more efficiently. In this work, we employed two different graph construction algorithms: the NN-descent proposed by Dong et al. [16] and the search-based insertion algorithm used by Malkov et al. [29]. The NN-descent is an iterative procedure initialized with randomly selected nearest neighbors. In each iteration, a random sample of queries is selected to participate in neighborhood propagation.

Malkov et al. [29] called their method a *Small World* (SW) graph. The graph-building algorithm finds an insertion point by running the same algorithm that is used during retrieval. Multiple insertion attempts are carried out starting from a random point.

The *open-source* implementation of NN-descent is publicly available online.[7]. However, it comes without a search algorithm. Thus, we used the algorithm due to Malkov et al. [29], which was available in the Non-Metric Space Library [8]. We applied both graph construction algorithms. Somewhat surprisingly, in all but two cases, NN-descent took (much) longer time to converge. For each data set, we used the graph-construction algorithm that performed better on a subset of the data. Both graph construction algorithms are computationally expensive and are, therefore, constructed in a multi-threaded mode (four threads). Tuning of k-NN graphs involved manual selection of two parameters k and the decay coefficient (tuning was carried out on a subset of data). The latter parameter, which is used only for NN-descent, defines the convergence speed.

Brute-force filtering is a simple approach where we exhaustively compare the permutation of the query against permutation of every data point. We then use incremental sorting to select γ permutations closest to the query permutation. These γ entries represent candidate records compared directly against the query using the original distance.

As noted in § 2, the cost of the filtering stage is high. Thus, we use this algorithm only for the computationally intensive distances: SQFD and the Normalized Levenshtein distance. Originally, both in the case of SQFD and Normalized Levenshtein distance, good performance was achieved with permutations of the size 128. However, Levenshtein distance was applied to DNA sequences, which were strings whose average length was only 32. Therefore, using uncompressed permutations of the size 128 was not space efficient (128 32-bit integers use 512 bytes). Fortunately, we can achieve the same performance using bit-packed binary permutations with 256 elements, each of which requires only 32 bytes.

The optimal permutation size was found by a small-scale grid search (again using a subset of data). Several values of γ (understood as a fraction of the total number of points) were manually selected to achieve recall in the range 0.85-0.9.

NAPP is a neighborhood approximation index described in § 2 [40]. Our implementation is different from the proposition of Chávez et al. [24] and Tellez et al. [39] in at least two ways: (1) we do not compress the index and (2) we use a simpler algorithm, namely, the ScanCount, to merge posting lists [13]. For each entry in the database, there is a counter. When we read a posting list entry corresponding to the object i, we increment counter i. To improve cache utilization and overall performance, we split the inverted index into small chunks, which are processed one after another. Before each search counters are zeroed using the function `memset` from a standard C library.

Tuning NAPP involves selection of three parameters m (the total number of pivots), m_i (the number of indexed pivots), and t. The latter is equal to the minimum number of indexed pivots that has to be shared between the query and a data point. By carrying out a small-scale grid search, we found that increasing m improves both recall and decreases retrieval time, yet, improvement is small beyond $m = 500$. At the same time, computation of one permutation entails computation of m distances to pivots. Thus, larger values of m incur higher indexing cost. Values of m between 500 and 2000 provide a good trade-off. Because the indexing algorithm is computationally expensive, it is executed in a multi-threaded mode (four threads).

Increasing m_i improves recall at the expense of retrieval efficiency: The larger is m_i, the more posting lists are to be processed at query time. We found that good results are achieved for $m_i = 32$. Smaller values of t result in high recall values. At the same time, they also produce a larger number of candidate records, which negatively affects performance. Thus, for cheap distances, e.g. L_2, we manually select the smallest t that allows one to achieve a desired recall (using a subset of data). For more expensive distances, we have an additional filtering step (as proposed by Tellez et al. [39]), which involves sorting by the number of commonly indexed

[7] `https://code.google.com/p/nndes/`

pivots.

Our initial assessment showed that NAPP was more efficient than the PP-index and at least as efficient MI-file, which agrees with results of Chávez et al. [11]. We also compared our NAPP implementation to that of Chávez et al. [11] using the same L_1 data set: 10^6 normalized CoPhIR descriptors. At 95% recall, Chávez et al. [11] achieve a 14x speed up, while we achieve a 15x speed up (relative to respective brute-force search implementations). Thus, our NAPP implementation is a competitive benchmark. Additionally we benchmark our own implementation of Fagin et al.'s OMEDRANK algorithm [20] and found NAPP to be more efficient. We also experimented with indexing permutations using the VP-tree, yet, this algorithm was either outperformed by the VP-tree in the original space or by NAPP.

3.3 Experimental Setup

Experiments were carried out on an Linux Intel Xeon server (3.60 GHz, 32GB memory) in a single threaded mode using the *Non-Metric Space Library* [8] as an evaluation toolkit. The code was written in C++ and compiled using GNU C++ 4.8 with the `-Ofast` optimization flag. Additionally, we used the flag `-march=native` to enable SIMD extensions.

We evaluated performance of a 10-NN search using a procedure similar to a five-fold cross validation. We carried out five iterations, in which a data set was randomly split into two parts. The larger part was indexed and the smaller part comprised queries [8]. For each split, we evaluated retrieval performance by measuring the average retrieval time, the improvement in efficiency (compared to a single-thread brute-force search), the recall, the index creation time, and the memory consumption. The retrieval time, recall, and the improvement in efficiency were aggregated over five splits. To simplify our presentation, in the case of non-symmetric KL-divergence, we report results only the for the left queries. Results for the right queries are similar.

Because we are interested in high-accuracy (near 0.9 recall) methods, we tried to tune parameters of the methods (using a subset of the data) so that their recall falls in the range 0.85-95. Method-specific tuning procedures are described in respective subsections of Section 3.2.

3.4 Quality of Permutation-Based Projections

Recall that permutation methods are filter-and-refine approaches that map data from the original space to L_2 or L_1. Their accuracy depends on the quality of this mapping, which we assess in this subsection. To this end, we explore (1) the relationship between the original distance values and corresponding values in the projected space, (2) the relationship between the recall and the fraction of permutations scanned in response to a query.

Figure 2 shows distance values in the original space (on the x-axis) vs. values in the projected space (on the y-axis) for eight combinations of data sets and distance functions. Points were randomly sampled from two strata: a complete subset and a set of points that are 100-nearest neighbors of randomly selected points. Of the presented panels, 2a and 2b correspond to the classic random projections. The remaining panels show permutation-based projections.

[8]For cheap distances (e.g., L_2) the query set has the size 1000, while for more expensive ones (such as the SQFD), we used 200 queries for each of the five splits.

Classic random projections are known to preserve inner products and distance values [5]. Indeed, the relationship between the distance in the original and the projected space appears to be approximately linear in panels 2a and 2b. Therefore, it preserves the relative distance-based order of points with respect to a query. For example, there is a high likelihood for the nearest neighbor in the original space to remain the nearest neighbor in the projected space. In principle, any monotonic relationship—not necessarily linear–will suffice [38]. If the monotonic relationship holds at least approximately, the projection typically distinguishes between points close to the query and points that are far away.

For example, the projection in panel 2e appears to be quite good, which is not entirely surprising, because the original space is Euclidean. The projections in panels 2h and 2d are also reasonable, but not as good as one in panel 2e. The quality of projections in panels 2f and 2c is somewhat uncertain. The projection in panel 2g–which represents the non-symmetric and non-metric distance–is obviously poor. Specifically, there are two clusters: one is close to the query (in the original distance) and the other is far away. However, in the projected space these clusters largely overlap.

Figure 3 contains nine panels that plot recall (on x-axis) against a fraction of candidate records necessary to retrieve to ensure this recall level (on y-axis). In each plot, there are several lines that represent projections of different dimensionality. Good projections (e.g., random projections in panels 3a and 3b) correspond to steep curves: recall approaches one even for a small fraction of candidate records retrieved. Steepness depends on the projection dimensionality. However, good projection curves are steep even in relatively low dimensions.

The worst projection according to Figure 2 is in panel 2g. It corresponds to the Wiki-128 data set with distance measured by KL-divergence. Panel 3f in Figure 3, corresponding to this combination of the distance and the data set, also confirms the low quality of the projection. For example, given a permutation of dimensionality 1024, scanning 1% of the candidate permutations achieves roughly a 0.9 recall. An even worse projection example is in panel 3e. In this case, regardless of the dimensionality, scanning 1% of the candidate permutations achieves recall below 0.6.

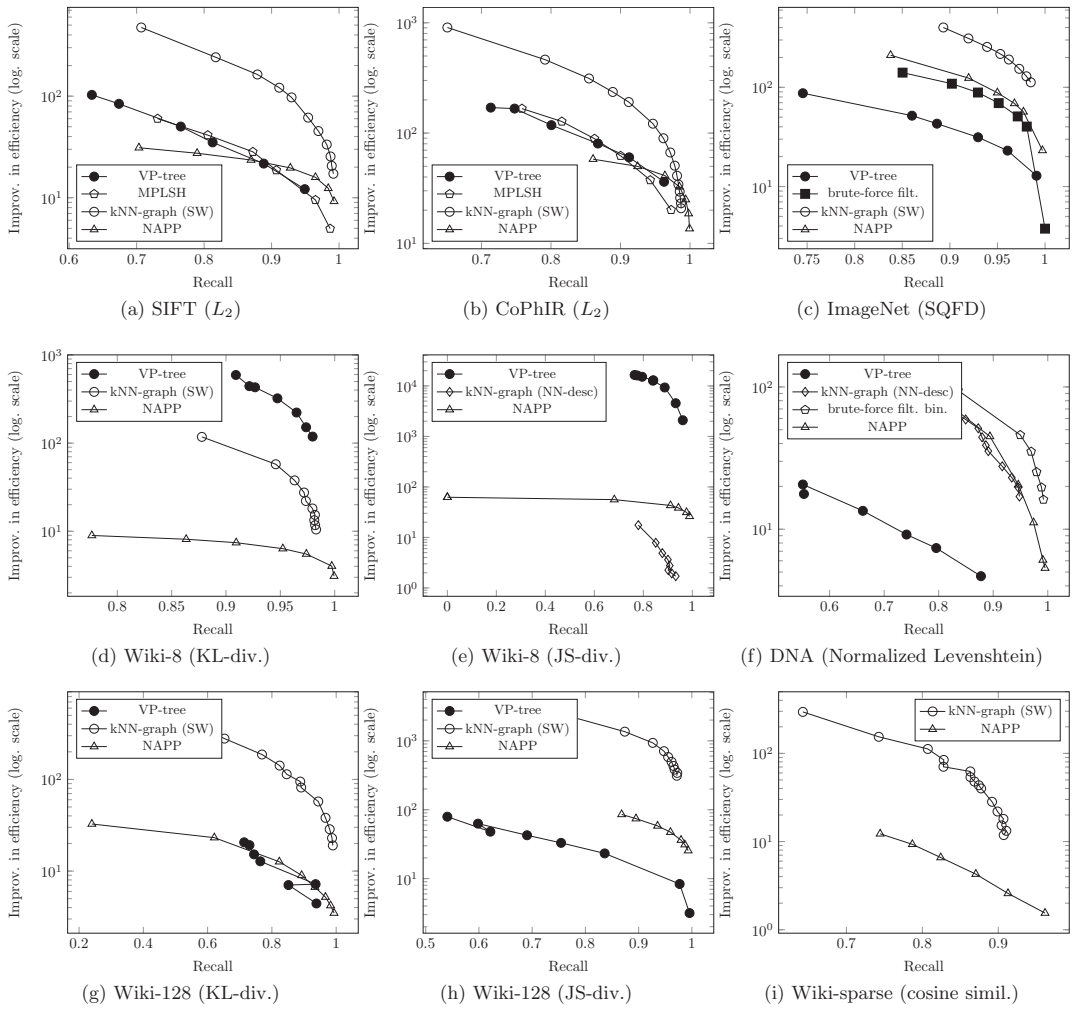
At the same time, for majority of projections in other panels, scanning 1% of the candidate permutations of dimensionality 1024 achieves an almost perfect recall. In other words, for some data sets, it is, indeed, possible in most cases to obtain a tiny set of candidate entries containing a true near-neighbor by searching in the permutation space.

3.5 Evaluation of Efficiency vs Recall

In this section, we use complete data sets listed in Table 1. Figure 4 shows nine data set specific panels with improvement in efficiency vs. recall. Each curve captures method-specific results with parameter settings tuned to achieve recall in the range of 0.85-0.95.

It can be seen that in most data sets the permutation method NAPP is a competitive baseline. In particular, panels 4a and 4b show NAPP outperforming the state-of-the art implementation of the multi-probe LSH (MPLSH) for recall larger than 0.95. This is consistent with findings of Chávez et al. [11].

In that, in our experiments, there was no single best method. k-NN graphs substantially outperform other meth-



(a) SIFT (L_2) (b) CoPhIR (L_2) (c) ImageNet (SQFD)

(d) Wiki-8 (KL-div.) (e) Wiki-8 (JS-div.) (f) DNA (Normalized Levenshtein)

(g) Wiki-128 (KL-div.) (h) Wiki-128 (JS-div.) (i) Wiki-sparse (cosine simil.)

Figure 4: Improvement in efficiency vs. recall for various data sets (10-NN search). Each plot includes one of the two implemented k-NN graph algorithms: Small World (SW) or NN-descent (NN-desc).

ods in 6 out of 9 data sets. However, in low-dimensional data sets shown in panels 4d and 4e, the VP-tree outperforms the other methods by a wide margin. The Wiki-sparse data set (see panel 4i), which has high representational dimensionality, is quite challenging. Among implemented methods, only k-NN graphs are efficient for this set.

Interestingly, the winner in panel 4f is a brute-force filtering using binarized permutations. Furthermore, the brute-force filtering is also quite competitive in panel 4c, where it is nearly as efficient as NAPP. In both cases, the distance function is computationally intensive and a more sophisticated permutation index does not offer a substantial advantage over a simple brute-force search in the permutation space.

Good performance of k-NN graphs comes at the expense of long indexing time. For example, it takes almost four hours to built the index for the Wiki-sparse data set using as many as four threads (see Table 2). In contrast, it takes only 8 minutes in the case of NAPP (also using four threads). In general, the indexing algorithm of k-NN graphs is substantially slower than the indexing algorithm of NAPP: it takes up to an order of magnitude longer to build a k-NN graph. One exception is the case of Wiki-128 where the distance is the JS-divergence. For both NAPP and k-NN graph, the indexing time is nearly 40 minutes. However, the k-NN graph retrieval is an order of magnitude more efficient.

Both NAPP and the brute-force searching of permutations have high indexing costs compared to the VP-tree. This cost is apparently dominated by time necessary to compute permutations. Recall that obtaining a permutation entails m distance computations. Thus, building an index entails $N \cdot m$ distance computations, where N is the number of data points. In contrast, building the VP-tree requires roughly $N \cdot \log_2 N/b$ distance computations, where b is the size of the bucket. In our setup, $m > 100$ while $\log_2 N/b < 20$. Therefore, the indexing step of permutation methods is typically much longer than that of the VP-tree.

Even though permutation methods may not be the best solutions when both data and the index are kept in main memory, they can be appealing in the case of disk-resident data [2] or data kept in a relational database. Indeed, as noted by Fagin et al. [20], indexes based on the inverted files are *database friendly*, because they require neither complex data structures nor many random accesses. [9] Furthermore, deletion and addition of records can be easily implemented. In that, it is rather challenging to implement a dynamic version of the VP-tree on top of a relational database.

We also found that all evaluated methods perform reasonably well in the surveyed non-metric spaces. This might indicate that there is some truth to the two folklore wisdoms: (1) "the closest neighbor of my closest neighbor is my neighbor as well", (2) "if one point is close to a pivot, but another is far away, such points cannot be close neighbors". Yet, these wisdoms are not universal. For example, they are violated in one dimensional space with the "distance" $e^{-|x-y|}|x-y|$. In this space, points 0 and 1 are distant. However, we can select a large positive number that can be arbitrarily close to both of them, which results in violation of both property (1) and (2).

It seems that such a paradox does not manifest in the surveyed non-metric spaces. In the case of continuous functions, there is non-negative strictly monotonic transformation $f(x) \geq 0$, $f(0) = 0$ such that $f(d(x, y))$ is a μ-defective distance function. Thus, the distance satisfies the following inequality:

$$|f(d(q,a)) - f(d(q,b))| \leq \mu f(d(a,b)), \mu > 0 \qquad (1)$$

Indeed, a monotonic transformation of the cosine similarity is the metric function (i.e, the angular distance) [42]. The square root of the JS-divergence is metric function called Jensen-Shannon distance [18]. The square root of all Bregman divergences (which include the KL-divergence) is μ-defective as well [1]. The normalized Levenshtein distance is a non-metric distance. However, for many realistic data sets, the triangle inequality is rarely violated. In particular, we verified that this is the case of our data set. The normalized Levenshtein distance is approximately metric and, thus, it is approximately μ-defective (with $\mu = 1$).

If Inequality (1) holds, due to properties of $f(x)$, $d(a, b) = 0$ and $d(q, a) = 0$ implies $d(q, b) = 0$. Similarly if $d(q, b) = 0$, but $d(q, a) \neq 0$, $d(a, b)$ cannot be zero either. Moreover, for a sufficiently large $d(q, a)$ and sufficiently small $d(q, b)$, $d(a, b)$ cannot be small. Thus, the two folklore wisdoms are true if the strictly monotonic distance transformation is μ-defective.

4. CONCLUSIONS

We benchmarked permutation methods for approximate k-nearest neighbor search for generic spaces where both data and indices are stored in main memory (aiming for high-accuracy retrieval). We found these filter-and-refine methods to be reasonably efficient. The best performance is achieved either by NAPP or by brute-force filtering of permutations. For example, NAPP can outperform the multi-probe LSH in L_2. However, permutation methods can be outstripped by either VP-trees or k-NN graphs, partly because the filtering stage can be costly.

We believe that permutation methods are most useful in non-metric spaces of moderate dimensionality when: (1) The distance function is expensive (or the data resides on disk); (2) The indexing costs of k-NN graphs are unacceptably high; (3) There is a need for a simple, but reasonably efficient, implementation that operates on top of a relational database.

5. ACKNOWLEDGMENTS

This work was partially supported by the iAd Center [10] and the Open Advancement of Question Answering Systems (OAQA) group [11].

We also gratefully acknowledge help of several people. In particular, we are thankful to Anna Belova for helping edit the experimental and concluding sections. We thank Christian Beecks for answering questions regarding the Signature Quadratic Form Distance (SQFD) [4]; Daniel Lemire for providing the implementation of the SIMD intersection algorithm; Giuseppe Amato and Eric S. Tellez for help with data sets; Lu Jiang[12] for the helpful discussion of image retrieval algorithms and for providing useful references.

We thank Nikita Avrelin and Alexander Ponomarenko for porting their proximity-graph based retrieval algorithm to

[9]The brute-force filtering of permutations is a simpler approach, which is also database friendly.

[10]http://www.iad-center.com/

[11]http://oaqa.github.io/

[12]http://www.cs.cmu.edu/~lujiang/

the Non-Metric Space Library[13]. The results of the preliminary evaluation were published elsewhere [32]. In the current publication, we use improved versions of the NAPP and baseline methods. In particular, we improved the tunning algorithm of the VP-tree and we added another implementation of the proximity-graph based retrieval [16]. Furthermore, we experimented with a more diverse collection of (mostly larger) data sets. In particular, because of this, we found that proximity-based retrieval may not be an optimal solution in all cases, e.g., when the distance function is expensive to compute.

6. REFERENCES

[1] A. Abdullah, J. Moeller, and S. Venkatasubramanian. Approximate bregman near neighbors in sublinear time: Beyond the triangle inequality. In *Proceedings of the twenty-eighth annual symposium on Computational geometry*, pages 31–40. ACM, 2012.

[2] G. Amato, C. Gennaro, and P. Savino. MI-file: using inverted files for scalable approximate similarity search. *Multimedia tools and applications*, 71(3):1333–1362, 2014.

[3] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd international conference on Scalable information systems*, InfoScale '08, pages 28:1–28:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[4] C. Beecks. *Distance based similarity models for content based multimedia retrieval*. PhD thesis, 2013.

[5] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250. ACM, 2001.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[7] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2, 2009.

[8] L. Boytsov and B. Naidan. Engineering efficient and effective non-metric space library. In *SISAP*, pages 280–293, 2013. Available at `https://github.com/searchivarius/NonMetricSpaceLib`

[9] L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *NIPS*, pages 1574–1582, 2013.

[10] L. Cayton. Fast nearest neighbor retrieval for bregman divergences. In *ICML*, pages 112–119, 2008.

[11] E. Chávez, M. Graff, G. Navarro, and E. Téllez. Near neighbor searching with k nearest references. *Information Systems*, 2015.

[12] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM computing surveys (CSUR)*, 33(3):273–321, 2001.

[13] L. Chen and X. Lian. Efficient similarity search in nonmetric spaces with local constant embedding. *Knowledge and Data Engineering, IEEE Transactions on*, 20(3):321–336, 2008.

[14] P. Diaconis. Group representations in probability and statistics. *Lecture Notes-Monograph Series*, pages i–192, 1988.

[15] W. Dong. *High-Dimensional Similarity Search for Large Datasets*. PhD thesis, Princeton University, 2011.

[16] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586. ACM, 2011.

[17] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 669–678, New York, NY, USA, 2008. ACM.

[18] D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003.

[19] A. Esuli. PP-index: Using permutation prefixes for efficient and scalable approximate similarity search. *Proceedings of LSDS-IR*, 2009, 2009.

[20] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 301–312, New York, NY, USA, 2003. ACM.

[21] A. Faragó, T. Linder, and G. Lugosi. Fast nearest-neighbor search in dissimilarity spaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(9):957–962, 1993.

[22] K. Figueroa and K. Fredriksson. Speeding up permutation based indexing with indexing. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, pages 107–114. IEEE Computer Society, 2009.

[23] K.-S. Goh, B. Li, and E. Chang. Dyndex: a dynamic and non-metric space indexer. In *Proceedings of the tenth ACM international conference on Multimedia*, pages 466–475. ACM, 2002.

[24] E. C. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(9):1647–1658, 2008.

[25] D. Jacobs, D. Weinshall, and Y. Gdalyahu. Classification with nonmetric distances: Image retrieval and class representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(6):583–600, 2000.

[26] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 861–864. IEEE, 2011.

[27] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi.

[13] `github.com/searchivarius/NonMetricSpaceLib`

H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[28] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen. Sk-lsh: An efficient index structure for approximate nearest neighbor search. *Proc. VLDB Endow.*, 7(9):745–756, May 2014.

[29] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[30] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Experimental Algorithms*, pages 85–97. Springer, 2006.

[31] V. Pestov. Indexability, concentration, and {VC} theory. *Journal of Discrete Algorithms*, 13(0):2 – 18, 2012. Best Papers from the 3rd International Conference on Similarity Search and Applications (SISAP 2010).

[32] A. Ponomarenko, N. Avrelin, B. Naidan, and L. Boytsov. Comparative analysis of data structures for approximate nearest neighbor search. In *DATA ANALYTICS 2014, The Third International Conference on Data Analytics*, pages 125–130, 2014.

[33] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge, 2014.

[35] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[36] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS@ VLDB*, pages 1–8, 2011.

[37] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, pages 291–296. IEEE, 2002.

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

[39] E. S. Téllez, E. Chávez, and A. Camarena-Ibarrola. A brief index for proximity searching. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 529–536. Springer, 2009.

[40] E. S. Tellez, E. Chávez, and G. Navarro. Succinct nearest neighbor search. *Information Systems*, 38(7):1019–1030, 2013.

[41] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, 40(4):175–179, 1991.

[42] S. Van Dongen and A. J. Enright. Metric distances derived from cosine similarity and pearson and spearman correlations. *arXiv preprint arXiv:1208.3145*, 2012.

[43] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 194–205. Morgan Kaufmann, August 1998.

[44] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321, 1993.

[45] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

Appendix B

Paper II: Bregman hyperplane trees for fast approximate nearest neighbor search

Bilegsaikhan Naidan and Magnus Lie Hetland.
Appeared at International Journal of Multimedia Data Engineering and Management (IJMDEM), 2012.

Is not included due to copyright

Appendix C

Paper III: Static-to-dynamic transformation for metric indexing structures (extended version)

Bilegsaikhan Naidan and Magnus Lie Hetland.
Appeared at the Elsevier Information Systems Journal, 2014.

Static-to-dynamic transformation for metric indexing structures (extended version)

Bilegsaikhan Naidan, Magnus Lie Hetland

Department of Computer and Information Science,
Norwegian University of Science and Technology,
Sem Sælands vei 7-9, NO-7491 Trondheim, Norway

Abstract

In this paper, we study the well-known algorithm of Bentley and Saxe in the context of similarity search in metric spaces. We apply the algorithm to existing static metric index structures, obtaining dynamic ones. We show that the overhead of the Bentley-Saxe method is quite low, and because it facilitates the dynamic use of any state-of-the-art static index method, we can achieve results comparable to, or even surpassing, existing dynamic methods. Another important contribution of our approach is that it is very simple—an important practical consideration. Rather than dealing with the complexities of dynamic tree structures, for example, the core index can be built statically, with full knowledge of its data set.

Keywords: similarity search, static and dynamic indexes, Bentley-Saxe algorithm, experiments.

1. Introduction

Many modern applications require efficient similarity retrieval, including applications in multimedia (to find similar images, audio in digital-repositories), pattern recognition (to identify finger-prints, face images in image databases), and string searching (to find words in a dictionary while permitting spelling errors). In such applications, the search problem is often stated in terms of distance-search in a metric space. That is, given a metric d over a universe $\mathbb{U}$, and a data set $\mathbb{D} \subset \mathbb{U}$, find the objects in $\mathbb{D}$ that are closest to some query $q \in \mathbb{U}$ (either all within a search radius r, or the k nearest neighbors, kNN).

Rather than performing a linear scan of the full data set, it is common to preprocess the data set by building an index structure, exploiting the metric axioms (the triangular inequality in particular). Most existing such index structures are *static*.[1] That is, the index is built with access to the full data set, and if an object is to be added or deleted, a full rebuild of the entire index is normally required. Such rebuilding is, of course, time consuming and computationally intensive. To accommodate insertions and deletions, some special-purpose *dynamic* index structures, supporting additions and deletions at low cost, have been proposed. Maintaining the integrity and performance of a dynamic structure over time, with only incremental information, can be challenging; such structures can be more complicated, as well as less able to utilize global information about the data set.

In this paper,[2] we study the Bentley-Saxe [1] algorithm in the context of similarity search in metric spaces. The Bentley-Saxe method is a tool that allows us to transform a static data structure into a dynamic one for any decomposable search problem (as explained in Section 3). This means that we can still use the state of the art in static indexing, even if we need the functionality of a dynamic indexing method, without losing the ability to globally analyze the data set, and without adding any appreciable complexity. In fact, the Bentley-Saxe method can use the indexing methods as black-box modules, permitting a clean separation of the (static)

Email addresses: bileg@idi.ntnu.no (Bilegsaikhan Naidan), mlh@idi.ntnu.no (Magnus Lie Hetland)

[1] Based on an analysis of the proceedings of the International Workshop on Similarity Search and Applications.

[2] An abbreviated version of this paper appeared in [11].

indexing and the dynamism.

This paper is organized as follows. Section 2 describes some related work. The Bentley-Saxe method is explained in Section 3. Section 4 provides our experimental results. Some concluding remarks are given in Section 5.

2. Related Work

In this section we briefly overview some relevant static and dynamic metric indexing structures. For further details, refer to the tutorial by Hetland [10] and the books by Zezula et al. [20] and Samet [16]. We consider two well-known static methods (the VP-tree and the SSS-tree) as well as three dynamic ones (EGNAT, the DSA-tree and the M-tree).

The vantage point (VP) tree [19] is a static balanced binary tree. The construction algorithm for the VP-tree first selects a representative object p (a so-called *vantage point*) from the data set $\mathbb{D}$ and computes the median m of the distances between p and the other objects in the data set. Then it divides the data set into two subsets $\mathbb{D}_1$ and $\mathbb{D}_2$, such that $\mathbb{D}_1 = \{x \in \mathbb{D} \mid x \neq p, d(p,x) \leq m\}$ and $\mathbb{D}_2 = \mathbb{D} \setminus (\mathbb{D}_1 \cup \{p\})$. The algorithm recursively builds left and right subtrees for $\mathbb{D}_1$ and $\mathbb{D}_2$, if they are not empty. A range query q with radius r is performed by recursively traversing the tree from the root to leaves. For each visited node, $d(q,p)$ is computed and p is reported if $d(q,p) \leq r$. It is necessary to traverse the left subtree only if $d(q,p) - r \leq m$, and, similarly, the right subtree only if $d(q,p) + r > m$.

There exists a dynamic version of the VP-tree [9]. We have not compared our method to this dynamic VP-tree, however, as it is not at all straightforward to implement correctly, and in some cases is still unable to avoid periodic reconstruction of subtrees or even of the entire tree.

Brisaboa et al. have proposed a static index structure called the Sparse Spatial Selection (SSS) tree [3], in which the first object in a data set is selected as the first cluster center and then the rest of the objects become new cluster centers if they are far enough away from all current centers (i.e., the minimum distance between the object and current cluster centers is greater than αM, where α is an user-defined parameter and M is the maximum distance between any two objects); otherwise, they are assigned to the cluster associated with the nearest center. The process is recursively applied to those clusters that have not yet fallen below a given size threshold.[3]

The Geometric Near-neighbor Access Tree (GNAT) [2] is a multiway static tree and is built as follows. First, a set of pivots are selected at random and then the rest of the objects are assigned to a region associated with the closest pivot. Examples of a GNAT are shown in Figure 2a, 2b. For each region, the minimum and maximum distances to the other regions' objects are kept for efficiently filtering out non-promising regions in the search, meaning that a region is discarded if the query ball does not intersects with this distance interval. The subtrees are recursively built for all regions associated with the pivots.

The M-tree [7] is a hierarchical dynamic metric ball tree that is built in a bottom-up manner like B-trees. The insertion algorithm starts from the root and moves toward the leaves by selecting nodes that are closer to the new object or that require a minimum enlargement of existing balls. The new object is finally inserted into a leaf node. This may cause the leaf to split (if the node capacity is exceeded), which may trigger splits in some of its ancestor nodes, possibly even the root. For a leaf node split, the covering radius of the split node is set to the distance from the center to the object furthest away, that is, the actual covering radius. For an internal node split, the covering radius is not computed exactly, but over-estimated, as follows. For every child node, the covering radius of that node is added to the distance between the center of that child and that of the split node. Then, the covering radius of the split node is then set to the maximum of those sums. Figure 1 shows an example of the M-tree. Ciaccia and Patella [6] developed a bulk loading method for the M-tree and it can reduce the build cost significantly. However, Zezula et al. claimed that the resulting tree provides only slightly better search performance than an M-tree with the incremental insertion method [20, p. 111].

The Evolutionary GNAT (EGNAT) [18] is a dynamic version of GNAT. The root is initially created as a leaf node. The insertion algorithm traverses the index structure by choosing the subtree associated with the closest pivot until a leaf node

[3] It could be argued that the SSS-tree is *almost* a dynamic structure, as one could build it incrementally by modifying the clustering method somewhat. It would, however, be impossible to find the space diameter *a priori* in such a scenario.

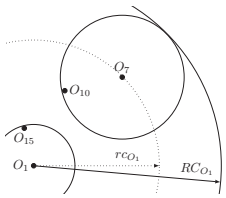


Figure 1: An example of the M-tree with two levels in $\mathbb{R}^2$ under $\mathcal{L}_2$. Top level with center O_1 and bottom levels with centers O_1 and O_7. RC_{O_1} represents the covering radius of top region centered at O_1 while rc_{O_1} represents its "true" (i.e., minimal) covering radius.

is reached. If the leaf node has room for the new object, it is added there. Otherwise, the leaf node is transformed into an internal node by selecting pivots and distributing its objects into new child (leaf) nodes. The leaf nodes also keep information about distances to their parent objects. During the search, this information is used to establish lower bounds to the actual distances between the query and objects.

The spatial approximation (SA) tree [12] is based on an approach that is, at least superficially, quite different from the hierarchical space decomposition of the other trees. First, an arbitrary object is selected as the root of the tree and a set of its neighbors is selected as follows. An object is inserted in the neighbor list if it is closer to the root than to all current neighbors. Otherwise, the object is assigned to a subset associated with its closest neighbor. Then, for each subset the procedure is applied recursively. Figure 2c shows an example of a SA-tree. The search algorithm uses a best-first branch-and-bound approach, similar to that used by most metric tree structures.

Navarro et al. [14] have shown that the SA-tree can be built dynamically, and they call the resulting structure the dynamic SA (DSA) tree. They manage to preserve the semantics of the SA-tree by introducing a *time-stamp* for every object. These time-stamps are then used during search, to ensure that only distance relationships that were known at the time of insertion are used when filtering out objects, to avoid false dismissals.

3. The Bentley-Saxe algorithm

We call a search problem *decomposable* if, for any pair of data sets $\mathbb{D}_1$ and $\mathbb{D} \setminus \mathbb{D}_1$, the answer to a query over $\mathbb{D}$ can be computed efficiently from the answers to queries for each of $\mathbb{D}_1$ and $\mathbb{D} \setminus \mathbb{D}_1$. The Bentley-Saxe algorithm (BS) exploits this sort of decomposition to reduce the size of the structures that need to be rebuilt, on average (i.e., amortized), when inserting or deleting objects.

The main data structure of BS is a set of $m = \lfloor \log_2 n \rfloor + 1$ buckets[4] $B_0, B_1, \ldots, B_{m-1}$ and each bucket B_i is either empty or a static data structure that contains a collection of 2^i objects. To insert a new object into the index, the algorithm follows the same principle that is used for incrementing a binary counter, where the ith bit denotes the absence or presence of a static index structure in the bucket B_i. The search is performed by accessing non-empty buckets and combining the results. Pseudocode for the transformation is given in Algorithm 1. Note that the search starts from B_{m-1} and proceeds to B_0. This is intentional, as it may improve the efficiency of kNN search by shrinking the covering radius of the current kNN candidate set as much as possible early on.

Algorithm 1 Static to dynamic transformation

```
1: function Init():
2:     B_0 ← null; m = 0

3: function Insert(x):
4:     D ← {x}
5:     Find minimum k such that B_k = null
6:     for i ← 0 to k − 1:
7:         D ← D ∪ Unbuild(B_i)
8:         B_i ← null
9:     B_k ← Build(D)
10:    if k = m:
11:        B_{m+1} ← null; m ← m + 1

12: function Query(q):
13:    ans ← ∅
14:    for i ← m − 1 downto 0:
15:        if B_i ≠ null:
16:            Search using q in B_i and update ans with results
17:    return ans
```

Let us consider an example where we insert a new object into the existing data structure. The example is illustrated in Figure 3.

Let the buckets $B_0, B_1, \ldots, B_{k+2}$ be non-empty. Thus, the first empty bucket is B_{k+3}. We build an index structure for bucket B_{k+3} containing the new object and all the objects stored in buckets $B_0, B_1, \ldots, B_{k+2}$. After building this structure, buckets $B_0, B_1, \ldots, B_{k+2}$ are nulled. Buckets B_{k+4} and upward are unchanged.

[4]For a dynamic index, the required number of buckets is, of course, unknown at the outset. The problem size n is the number of objects added *so far*.

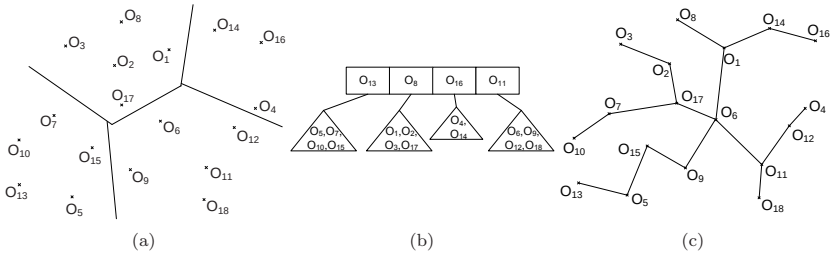


Figure 2: Examples of (a) a GNAT space decomposition with hyperplanes between O_8, O_{11}, O_{13} and O_{16}, (b) the corresponding GNAT tree, and (c) a SA-tree with the root O_6.

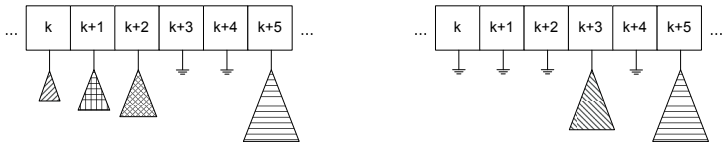


Figure 3: Illustrations of an index structure before the insertion of a new object (left) and after the insertion (right).

Now consider the asymptotic running time and space requirements of this approach. Let T be a static metric index structure with size $S_T(n)$ that can be constructed in time $C_T(n)$ and perform a query in time $Q_T(n)$. BS gives us a dynamic metric index structure T' based on T that requires the storage $S_{T'}(n) \in \mathcal{O}(S_T(n))$ and bulk construction time $C_{T'}(n) \in \mathcal{O}(C_T(n))$ (assuming that both storage and construction requirements for T are at least linear), and, because each object is inserted in $\log n$ buckets, an amortized insertion time of $I_{T'}(n) \in \mathcal{O}(\log n \cdot C_T(n)/n)$. In fact, if $C_T(n) \in \Omega(n^{1+\epsilon})$, for some $\epsilon > 0$, we have $I_{T'}(n) \in \mathcal{O}(C_T(n)/n)$, that is, there is no asymptotic overhead.[5] We can, in general, guarantee a query time of $\mathcal{O}(\log n \cdot Q_T(n))$. Moreover, if $Q_T(n) \in \Omega(n^\alpha)$ for some $\alpha > 0$, which is generally assumed [13], we can derive the even stronger bound $Q_{T'}(n) \in \mathcal{O}(Q_T(n))$. In other words, under reasonable assumptions for how static metric indexes work, we can quite simply construct dynamic versions with no asymptotic overhead.

The original version of BS method was not designed to handle deletions efficiently. Consider, for example, the scenario where we have a single non-empty bucket B_k, containing 2^k objects. To delete an object now, we have to split B_k into $B_0, B_1, \ldots, B_{k-1}$. This entails building k index structures, which might be prohibitively expensive. To address this, Overmars et al. [15] weakened the condition of the BS method so that every bucket B_k can be either empty or a static data structure which stores at least 2^{k-2} and at most 2^k objects. With this new condition, our deletion would affect only B_{k-2}, B_{k-1} and B_k. The approach of Overmars et al. is shown in Algorithm 2.

In line 7, we mark o as deleted in B_k. The bucket B_k might not be rebuilt until its total number of objects becomes 2^{k-2}. That would, of course, affect the search performance.[6] In order to decrease this effect, we introduce a parameter tuning option between lines 8 and 9. There are many possibilities for the parameter tuning. For instance, the bucket B_k can be rebuilt each time when 2^{k-3} objects have been deleted from that bucket. This is the strategy that is tested in our experiments.

[5] This can also be made to hold in the worst case, and not just amortized, using lazy rebuilding techniques that we have not studied in this paper.

[6] Note that a kNN search must still traverse the deleted objects, but they will not shrink the radius. So, in some cases, query performance an actually degrade as objects are deleted.

Algorithm 2 Overmars and Leeuwen

```
1: function Insert(x):
2:     Replace line 9 of Insert function of Algorithm 1 with the following
         if |D| > 2^{k-1}: B_k ← Build(D)
         else: B_{k-1} ← Build(D)                          ▷ |D| > 2^{k-2}

3: function Remove(o):
4:     Perform a range search with radius of 0 in B_k to find k such that o ∈ B_k
                                                  ▷ k from m − 1 downto 0
5:     if not found o:
6:         return false
7:     Delete o from B_k                          ▷ |B_k| is decremented by 1
8:     if |B_k| > 2^{k-2}:
9:         return true
10:    elif |B_k| = 2^{k-2} and k ≥ 2:
11:        if B_{k-1} ≠ null:
12:            D ← Unbuild(B_k) ∪ Unbuild(B_{k-1})
13:            if |B_{k-1}| > 2^{k-2}:
14:                B_{k-1} ← null
15:                B_k ← Build(D)
16:            else:
17:                B_k ← null
18:                B_{k-1} ← Build(D)
19:        elif B_{k-1} = null and B_{k-2} ≠ null:
20:            D ← Unbuild(B_k) ∪ Unbuild(B_{k-2})
                                          ▷ |B_k| + |B_{k-2}| > 2^{k-2}
21:            B_k ← null; B_{k-2} ← null
22:            B_{k-1} ← Build(D)
23:        elif B_{k-1} = null and B_{k-2} = null:
24:            D ← Unbuild(B_k); B_k ← null
25:            B_{k-2} ← Build(D)
26:        return true
```

4. Experiments

In this section we present our experimental evaluation of two new dynamic trees based on BS, comparing them against three existing dynamic trees. As the performance measure we used the number of distance computations required to construct index structures and to answer similarity queries. We have also investigated the overhead of the BS method, by comparing the build and search times of the static indexes to those of their transformed, dynamic counterparts. We have provided performance comparisons of range and kNN queries, as well as deletion costs per object and search performance after deletions.

4.1. The testbed

We performed experiments using both synthetic data sets, generated by us, and real-world data sets obtained from the SISAP metric space library [8]. For all vectors we use the Euclidean distance.

- Uniform 10: Synthetic. 100 000 uniformly generated 10-dimensional vectors.[7]
- Clusters 10: Synthetic. 100 000 clustered 10-dimensional vectors with 10 cluster centers. The centers were randomly chosen from a uniform distribution and objects in the clusters were generated from the multivariate normal distribution around each of the cluster centers with a variance of 0.1.
- Uniform 20: Synthetic. 100 000 uniformly generated 20-dimensional vectors.
- Clusters 20: Synthetic. 100 000 clustered 20-dimensional vectors with 100 cluster centers. We followed the same procedure as in Clusters 10 to generate the cluster centers.
- NASA: 40 150 feature vectors with 20 dimensions extracted from NASA images.
- Dictionary: a dictionary of 69 069 English words. Here we use the edit distance (or Levenstein distance), that is, the minimum number of insertions, deletions, and substitutions needed to transform one string into another.
- Histogram: a collection of 112 682 color histograms (112-dimensional vectors) from an image database.

Table 1 shows the intrinsic dimensionalities (*idim*s) [4] of the data sets. The distance histograms of the data sets are shown in Figure 4.

4.2. Experiment settings

We have applied the BS method to VP- and SSS-trees and call the resulting dynamic structures the BS-VP-tree and BS-SSS-tree, respectively. We have compared their performances to three dynamic metric index structures, the DSA-tree, EGNAT and M-tree. We set the maximum node fanout of the BS-SSS-tree to 5, 10, 20, 40 and 80. The parameter α was 0.45 for the 20-dimensional and 0.40 for the remaining of the data sets. The value of M is estimated before every (re)construction of a bucket as follows. An arbitrary object in the bucket is selected as the boundary object. Then, the distances between the boundary and all objects in the bucket are computed 10 times by maximizing the value of M and renewing the boundary object from current one. The cost of this estimation is also included in the construction and deletion costs. We used the SISAP implementation [8] of DSA-tree with time-stamping and bounded arity. The original authors [14, § 5.8] suggested that this version of the

[7] We also have 8 192 000 uniformly generated 10-dimensional vectors for the complexity analysis of the BS index.

Uniform 10	Clusters 10	Uniform 20	Clusters 20	NASA	Dictionary	Histogram
13.36	9.24	27.64	20.44	5.18	8.49	2.74

Table 1: *idim*s of the data sets.

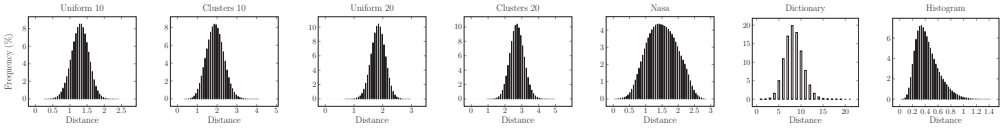


Figure 4: Distance distribution histograms.

DSA-tree would give the best results in terms of construction cost and search efficiency. The maximum arities of DSA-tree were set to 2, 4, 8, 16 and 32, as in their experiments. For EGNAT, we set the parameters by trial and error. We used internal node sizes of 4, 8, 12, 16 and 20 and maximum leaf node arities of 5, 10, 20, 40 and 80. In total, we performed 18 ($5 + 4 + 3 \times 3$) runs (with several queries).[8] For the M-tree, we used node sizes of 5, 10, 20, 40 and 80, the node split policy was mM_RAD and the method for distribution of objects was generalized hyperplane.

We randomly shuffled the order of all objects in each data set 10 times, obtaining 10 versions of the data set, and the results were averaged over 10 runs using these versions. For each run, a query set consists of 1000 queries which were selected randomly from the respective data set and the remaining objects in the data set used for indexing. We selected search radii for range queries so that we captured on average 0.01 %, 0.1 % and 1 % of the vectors. The search radii were in the range from 1 to 4 for the dictionary, capturing on average 0.003 %, 0.042 %, 0.361 %, and 1.946 % of the data set, respectively. For kNN search, we compared the search efficiency of the five structures by varying the result size thresholds, using the values $k = 1$, 5, 10, 20, 40 and 80. We report only best results in terms of search efficiency from the results obtained with different parameters use on every query set. The node sizes of the corresponding index that achieved the best search performance are listed in Table 2.

For deletions, the deletion cost includes both locating the object to be deleted (with a range search with radius 0) and the cost needed for deletion of the object from the index. The latter cost will be rebuilding buckets for BS-index. First, we constructed the BS-VP-tree, BS-SSS-tree and DSA-tree on the data sets. Then, we deleted every 10 % (randomly selected) of the corresponding data sets from the BS indexes and DSA-tree and then obtained the number of distance computations required to answer a query set in the BS indexes and DSA-tree.

Data set	EGNAT		DSA-tree	M-tree
	internal	leaf		
Uniform 10	4	5	4	40
Clusters 10	8	10	4	40
Uniform 20	20	20	32	20
Clusters 20	20	20	32	40
NASA	4	5	4	40
Dictionary	12	20	32	80
Histogram	8	10	4	40

Table 2: The node sizes of the indexes

We can tune the trade-off between deletion cost and search performance of DSA-tree by setting the fraction (percentage) of ghost hyperplanes permitted after deletions. We tested two versions of the DSA-tree. In the first, no ghost hyperplanes are permitted (all such hyperplanes produced during deletion are discarded by reconstruction), making deletion more expensive but search more efficient. In the second, all ghost hyperplanes are retained,

[8] Note that leaf node size should be greater than or equal to internal node size.

and deletion does not trigger any reconstruction. This makes the deletion cheaper, but search costs suffer. We call the two versions DSA-tree$_0$ and DSA-tree$_1$, respectively.

We implemented our experimental framework in C++, which was compiled in gcc 4.6.2 with the option -O3. All experiments are performed on a PC with a 3.3 GHz Intel Core i5-2500 processor and 8 GB RAM. We did not use any caching of distances during index construction and query processing.

4.3. The overhead of the BS index

First, let us consider the *construction cost* overhead of BS-based index structures. We constructed the VP-tree, SSS-tree (with maximum node fanout 5), BS-VP-tree and BS-SSS-tree (with maximum node fanout 5) 10 times on every 10 % of several data sets and obtained the ratio between the number of distance computations required to build the BS index and static index with the same settings, with the BS index built incrementally. The geometric mean of the ratio was 3.16 ± 1.23, with minimum and maximum values of 1.69 and 4.27. So in our experiments, the BS index is at most 4.27 times as costly to build as the corresponding static index. Figure 5 shows the construction cost overhead of the BS-based index structures.[9]

The figures show that the average ratio for the VP-tree is much higher than for the SSS-tree, and the values for VP-tree are distributed almost evenly. The values for SSS-tree are positively skewed in general, i.e., it has relatively few high values; it also performed particularly well on the dictionary.

Now let us consider the ratio for *index construction time* and *query set execution time* with all k and search radii. We followed the same principle previously used for the construction cost ratio to obtain these ratios with $2^m - 1$ objects for each data set. The motivation for this was to force all the buckets in the BS structure to be non-empty; that is, we intentionally increased the overhead of BS-based index structures, intending to elicit the worst-case search performance. The ratios are shown in Figure 6. For the construction time ratio, the maximum value for BS-VP-tree was 3.62 (with construction time 1.23 s) on the dictionary (Figure 6a) while the maximum value for the BS-SSS-tree was

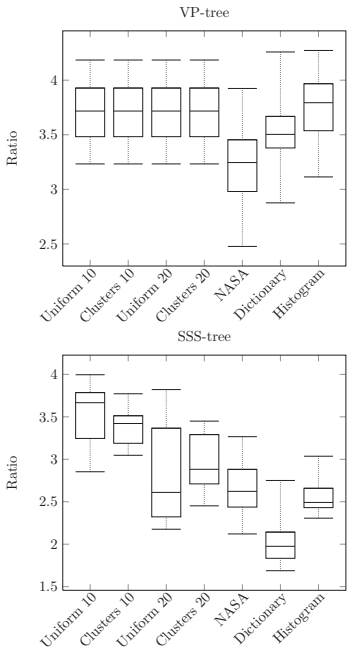


Figure 5: Construction cost ratio of BS index to static index with the same settings.

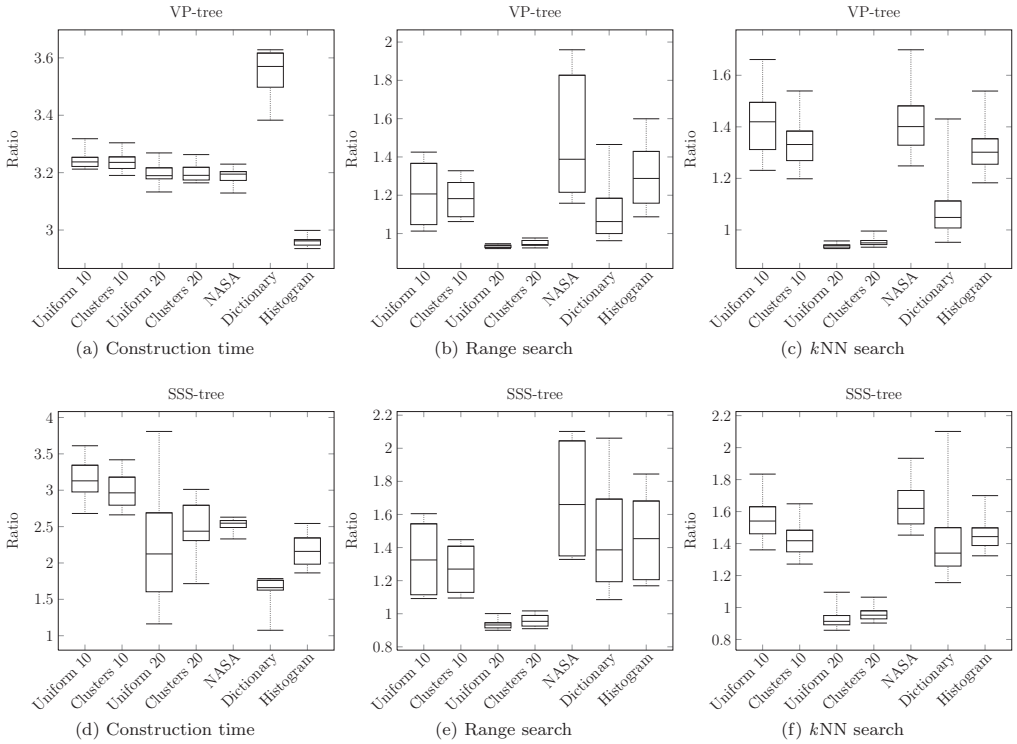
3.80 (with construction time 11.09 s) on Uniform 20 (Figure 6d). In Figure 6b, 6c, 6e and 6f, we see that there is almost no search time difference between static and BS-based index structures on the 20-dimensional synthetic vectors due to high *idim*s. Across all of our experiments, the maximum value of query set execution time for the static index structures was 19.87 s while for the BS-based index structures the maximum value was 20.97 s.

4.4. Comparison of construction costs

All index structures were built in an incremental fashion, i.e., initially all of the index structures were empty and then all objects in the data sets were added into the index one by one. The construction costs are shown in Table 3.

As the *idim* of the synthetic data sets increases we see that the data sets become difficult to index. This increase clearly affects the construction cost of the SSS-tree. This effect may be due to the fact that every object tends to become a cluster center of the tree because all objects are approximately equidis-

[9] These are standard box plots, showing the minimum, the 25 % quantile, the median, the 75 % quantile and the maximum value.



(a) Construction time (b) Range search (c) kNN search

(d) Construction time (e) Range search (f) kNN search

Figure 6: Construction time and query set execution time ratio of BS index to static index with same settings.

Data set	BS-VP-tree		BS-SSS-tree		EGNAT	DSA-tree	M-tree
	incremental	bulk	incremental	bulk			
Uniform 10	5 762 240	1 347 913	59 199 773	13 836 204	3 451 333	3 126 671	8 795 290
Clusters 10	5 762 240	1 347 913	38 317 600	9 429 190	5 327 960	3 298 234	8 777 797
Uniform 20	5 762 240	1 347 913	208 912 258	68 686 142	7 582 301	8 631 551	5 973 374
Clusters 20	5 762 240	1 347 913	96 468 450	26 626 524	8 876 612	8 857 394	8 782 329
NASA	1 952 946	486 453	13 619 821	3 983 649	1 646 678	1 219 949	3 269 566
Dictionary	4 676 487	1 079 805	90 720 799	31 105 813	4 820 213	5 531 384	8 734 896
Histogram	6 246 315	1 486 021	45 558 887	14 712 440	9 688 228	4 124 772	10 103 325

Table 3: Construction costs of index structures on various data sets.

tant from each other in high-dimensional spaces. It should also be noted that the clustering cost of the SSS-tree is high also in the static case, so this is not an artifact of our approach.

When considering any overhead in construction, it is important to note that our method is quite amenable to *bulk loading* (as shown in Table 3): If a given data set is available at the outset, or if a large number of objects are added, there is *no need* to build the structure incrementally, by adding individual objects. Instead, which buckets need to be filled can be easily calculated from the total data size, and the objects can be partitioned among these (e.g., randomly), and the static structures built. This means that there would be no need for multiple rebuilds, and the overhead would be much lower. For example, if the data size were a power of 2, there would be *no overhead whatsoever*. The resulting data structure would still retain all its dynamic properties. (The overhead in general will, of course, vary with how close the data size is to a power of 2, either above or below.)

We compared the memory usage of the BS-indexes with the DSA-tree which is given in Table 4. The results show that the BS-indexes require almost same amount of memory as the DSA-tree.

Data set	BS-VP-tree	BS-SSS-tree	DSA-tree
Uniform 10	17.02	17.26	19.62
Clusters 10	17.02	17.39	19.20
Uniform 20	21.70	23.19	28.00
Clusters 20	21.58	22.41	27.64
NASA	8.28	8.73	14.72
Dictionary	10.44	10.03	10.39
Histogram	62.98	64.55	62.44

Table 4: Memory usage for contruction of the indexes (MB).

4.5. Empirical complexity analysis of the BS index

We have now looked at the overhead of the BS algorithm beyond the corresponding static structures, and we have compared the construction costs of the BS-based structures and custom-designed dynamic ones. We now wish to tentatively examine the asymptotic complexity of the method, both for construction and search. We have some expectations about how the method ought to behave, but these are based on certain assumptions (primarily the running times of the static, underlying structures), which may not hold in practice.

To map out the functional relationship between input size and performance we use a doubling experiment [17]. To get a sufficiently large data set, we generated 8 192 000 uniform 10-dimensional vectors. We then measured performance with problem sizes at powers of 2, starting at 8000, with each experiment performed 10 times on 10 randomly shuffled versions of the data. In each experiment, we built the VP-tree, SSS-tree, BS-VP-tree and BS-SSS-tree[10] and obtained the ratio between the time required to build the BS indexes and the static indexes.

As discussed in Section 3, if the static build time were $\Omega(n^{1+\epsilon})$, for some $\epsilon > 0$, we would expect the ratio between the build-times for the static and BS-based dynamic structures to be a constant (no asymptotic overhead). For the VP-tree, however, the construction cost is $\Theta(n \log n)$ [see, e.g., 5], which means we would expect a log-factor between the static and incremental dynamic construction cost. While an asymptotic analysis is harder to do for the SSS-tree, because the arity varies from node to node, it is not unreasonable to expect a similar complexity. To explore this question, we have studied the normalized construction cost for the SSS-tree in isolation. In Figure 7, we have plotted $C_T(n)/n$ for the SSS-tree, with a logarithmic horizontal axis. A straight line would indicate a growth proportional to $n \log n$. The regression line has been included, and, as can be seen, $C_T(n)$ seems to grow more slowly than this (i.e., a sublinear trend in the log-plot), which would mean that we could expect the ratio of the static and BS-based build costs to be logarithmic here as well.

To see whether the ratios indeed *are* logarithmic, we have plotted them in a similar manner in Figure 8 (once again with a regression line), where a straight line would indicate a perfect logarithmic relationship between data size and the index build slowdown due to the BS method. It seems like the trend is, indeed, approximately linear.

We also performed kNN searches on the four structures by using all of the result size thresholds and obtained (i) the time required to answer a query set in the BS-based and static indexes and (ii) the ratio between the query set execution time in the BS indexes and in the static indexes. The

[10] The BS indexes were built in an incremental fashion.

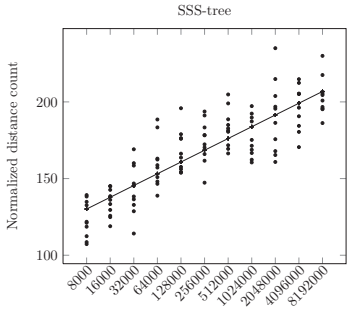


Figure 7: Normalized construction cost ($C_T(n)/n$) vs data set sizes on the synthetic set with SSS-trees. The normalized cost is seems sublinear.

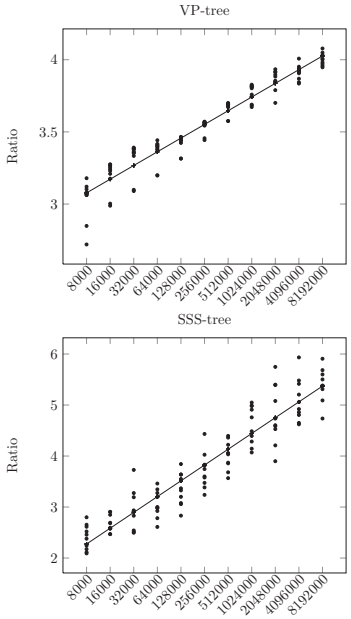


Figure 8: Construction time ratio of BS index to static index with same settings on various data set sizes.

results are given in Figure 9. (Note that both axes are logarithmic.) In this case, the expected ratios would depend on whether the query time is, indeed, $\Omega(n^\alpha)$, for some $\alpha > 0$. If this were the case, we would expect straight lines in subfigures (a) and (c), and we would have a constant performance ratio, which would show as a horizontal line in subfigures (b) and (d). As can be clearly seen, this is not exactly the case, although it may not be too far from the truth. It would seem that the empirical query performance (for the static structures) is somewhere between polylogarithmic and polynomial, leading to a ratio that grows, albeit slowly, with problem size.

4.6. Query performance, with and without deletions

Figure 10 shows the search results over the synthetic data sets and the impact of dimensionality. The performance of all of the index structures degrades when the dimensionality increases, especially for EGNAT and BS-VP-tree. In Figure 11, the search results over the real-world data sets are shown. The BS-VP-tree outperforms EGNAT for the real-world data sets and is comparable to DSA-tree and M-tree for range queries with low selectivity. For the dictionary, the BS-SSS-tree outperforms the DSA-tree with up to twice the search efficiency. In fact, in *all* of the experiments, the BS-SSS-tree outperforms all the other index structures in our experiments, a result almost certainly due to the efficiency of the SSS-tree itself, which comes at the price of a higher building cost. The contribution of our method in this case is that such a tradeoff between build cost and search efficiency *can be made in the first place*, by providing a dynamic version of the SSS-tree.

Deletions were performed on several data sets. After deletions, we performed range and 10 NN queries. The search radii for range search were selected so that we retrieve on average 0.1 % of the vectors and was set to 2 for the dictionary. We measured the distance computations (explained in Section 4.2) over several data sets. The results are shown in Figure 12. Each point in the figures represents an average. The highest deletion cost was 5182 with the DSA-tree$_0$ and it occured after deleting 10 % of the dictionary. The deletion cost of the BS-VP-tree is quite low in all of our experiments. The BS-SSS-tree achieved better search performance than the two competing indexes. For the dictionary, the deletion costs of the DSA-tree$_0$ are 10–16 times as high those of the BS-VP-tree, while

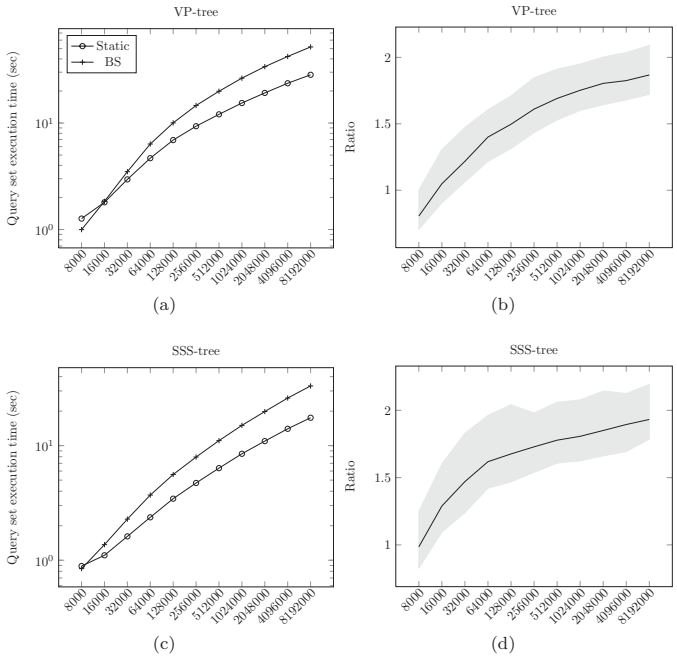


Figure 9: Mean of the query set execution time (a, c) and query set execution time ratio of BS index to static index with same settings (b, d) on various data set sizes. The gray area indicates the full range of values, whereas the line represents the mean of the values.

the DSA-tree$_0$ is about 1.3 times as fast as the BS-VP-tree. The deletion costs of the DSA-tree$_1$ are smaller than the DSA-tree$_0$ and the BS-SSS-tree. Its search performance, however, is no better. We also see that the deletion costs of the DSA-tree$_1$ is increased after deletions due to the presence ghost hyperplanes. In general, the deletion cost of the BS-index will increase if we rebuild the buckets several times. This yields some peaks in Figure 12. For the BS-SSS-tree, we analyzed this effect after deleting 50 % of NASA. The deletion cost was quite low (159) with respect to its previous and next points and there was no rebuilding of buckets. This means that some objects were marked as deleted in the index, and this should affect the search performance. However, the BS-indexes outperformed the DSA-tree for this deletion percentage.

Another observation we made was that the DSA-tree is very sensitive to the order of deletions. The deletion cost is very high if we delete the objects in the exact same order as they were inserted, whereas the cost is quite low if we delete them in the opposite order. For example, the deletion cost is 431 203 when deleting 10 % of the NASA data set in insertion order, while the cost is just 70 in the opposite order. No such effect is observed for BS-indexes.

5. Conclusions

We have studied the Bentley-Saxe algorithm for static-to-dynamic data structure transformations and how it can be applied to similarity search, yielding a simple method for transforming static index structures into dynamic ones. We have also empirically demonstrated that the method has a reasonably low overhead, both in terms of building and search cost. In fact, this overhead is low enough that when adapting a particularly efficient static data structure such as the SSS-tree, we can still achieve search times lower than comparable custom-designed dynamic data structures. In addition to this increased performance, the dynamic

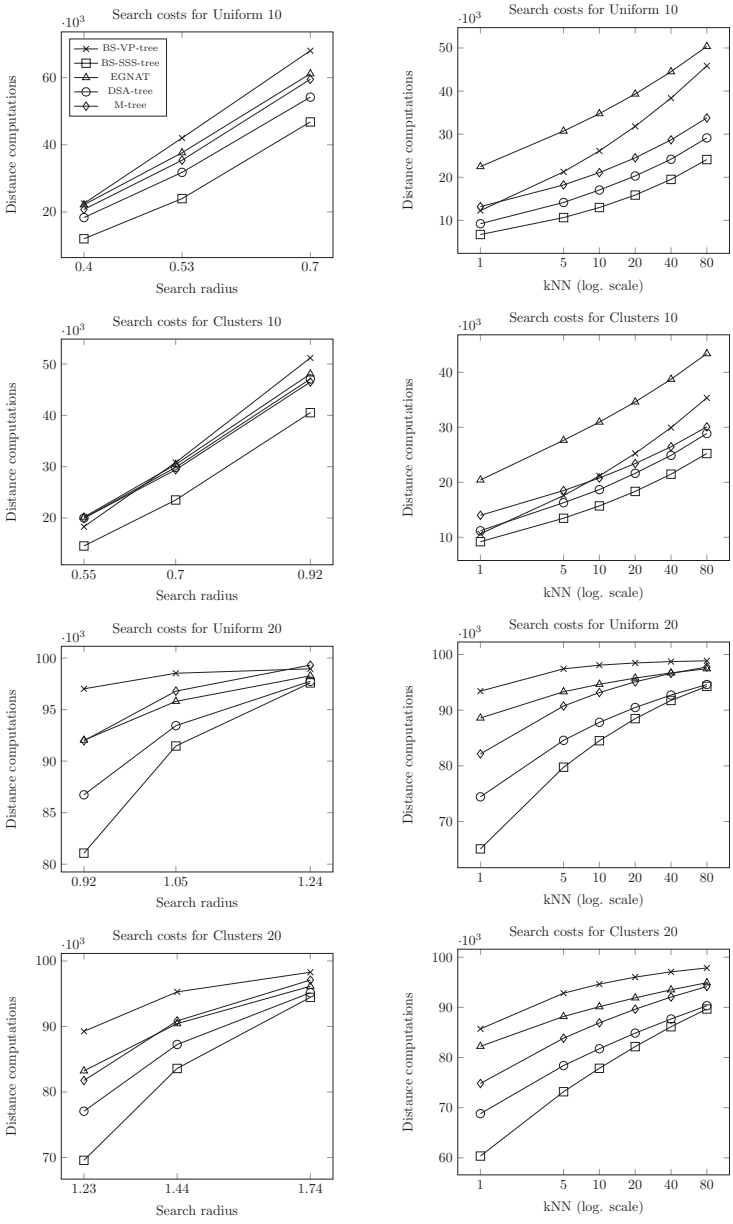


Figure 10: Performance evaluations on the synthetic data sets.

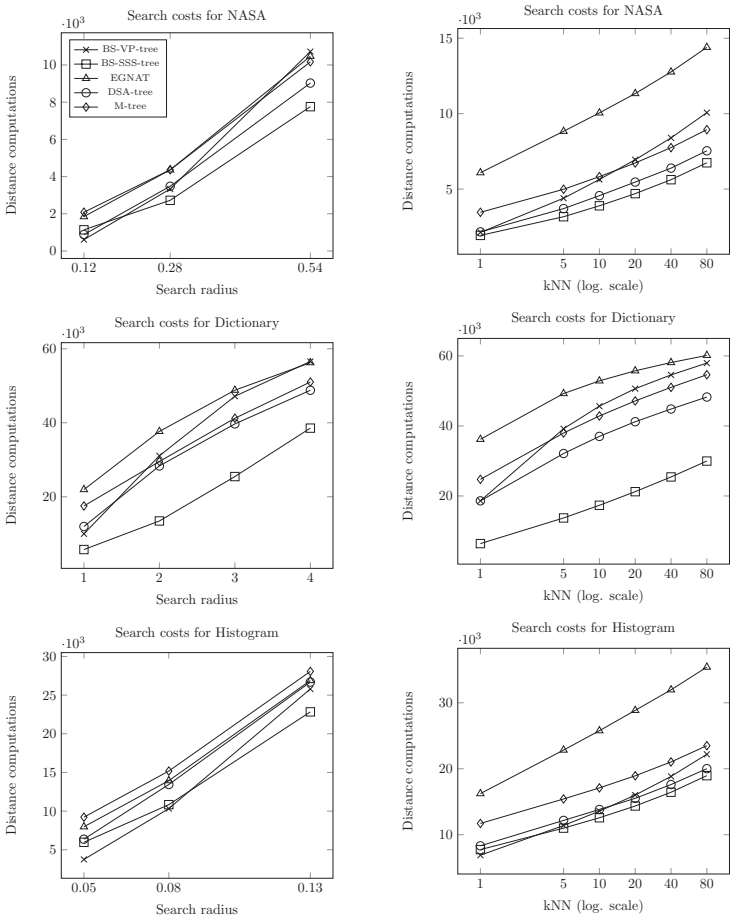


Figure 11: Performance evaluations on the real-world data sets.

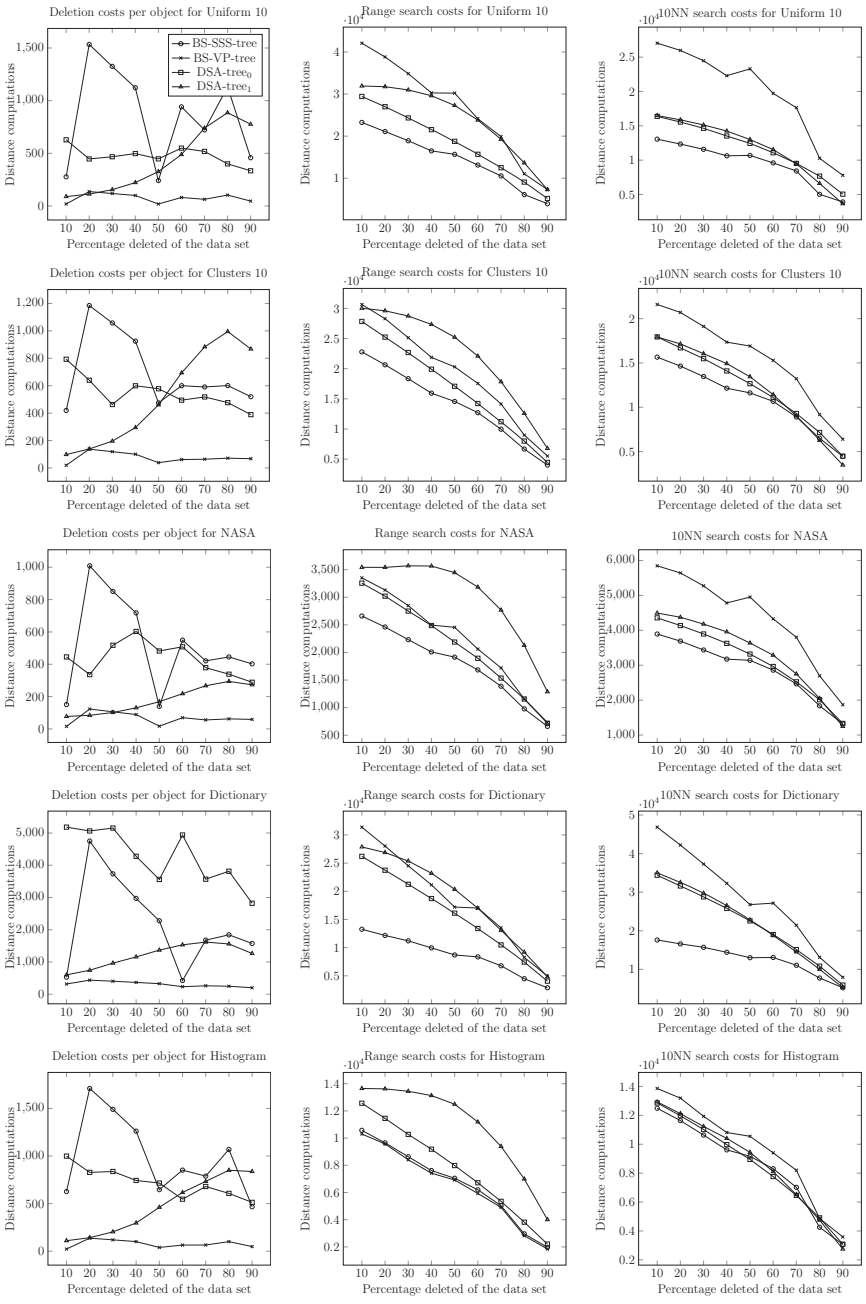


Figure 12: Deletion costs and performance evaluations after deletions on various data sets.

structures resulting from using the Bentley-Saxe method can be considerably less complex than other dynamic indexes, given that it is simply an isolated add-on to existing, usually simpler, static indexes.

There are still avenues of research that might be pursued, related to this topic. For example, more existing structures (static and dynamic) could be compared, and other parameter settings could be tried (for example, the number of deletions needed to trigger a rebuild). It might be possible to create hybrid structures, where dynamic methods are combined with the Bentley-Saxe idea. This could be useful, for example, for structures that are suitable for bulk construction, whose quality deteriorates with incremental modification. Such a combined approach might reduce some of the overhead inherent in our method, while still reaping some of its benefits.

References

[1] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301 – 358, 1980.

[2] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of 21th International Conference on Very Large Data Bases, VLDB*, pages 574–584, 1995.

[3] N. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *Proceedings of SOFSEM'08*, number 4910 in LNCS, pages 186–197, 2008.

[4] E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *Revised Papers from ALENEX'01*, pages 147–160, 2001.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[6] P. Ciaccia and M. Patella. Bulk loading the m-tree. In *In Proceedings of the 9th Australasian Database Conference (ADC'98)*, pages 15–26, 1998.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB*, pages 426–435. Morgan Kaufmann, 1997.

[8] K. Figueroa, G. Navarro, and E. Chavez. Metric spaces library, 2010. Downloaded February 20th, 2013 from `http://www.sisap.org/Metric_Space_Library.html`.

[9] A. W.-C. Fu, P. M.-S. Chan, Y.-L. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9(2):154–173, 2000.

[10] M. L. Hetland. The basic principles of metric indexing. In *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*. 2009.

[11] B. Naidan and M. L. Hetland. Static-to-dynamic transformation for metric indexing structures. *Lecture Notes in Computer Science*, 7404:101–115, 2012.

[12] G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, August 2002.

[13] G. Navarro. Analyzing metric space indexes: What for? In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, 2009.

[14] G. Navarro and N. Reyes. Dynamic spatial approximation trees. *Journal of Experimental Algorithmics (JEA)*, 12:1.5:1–1.5:68, 2008.

[15] M. Overmars and J. Leeuwen. Two general methods for dynamizing decomposable searching problems. *Computing*, 26:155–166, 1981.

[16] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[17] R. Sedgewick. *Algorithms in Java, Third Edition*. Addison-Wesley, 2003.

[18] R. Uribe, G. Navarro, R. J. Barrientos, and M. Marín. An index data structure for searching in metric space databases. In *Proceedings of the 6th International Conference of Computational Science, ICCS*, volume 3991, pages 611–617, 2006.

[19] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual Symposium on Discrete algorithms*, pages 311–321, 1993.

[20] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search : The Metric Space Approach*. Springer, 2006.

Paper IV: Learning to Prune in Metric and Non-Metric Spaces

Leonid Boytsov and Bilegsaikhan Naidan.
Appeared at the Neural Information Processing Systems (NIPS), 2013.

Appendix D

Learning to Prune in Metric and Non-Metric Spaces

Leonid Boytsov
Carnegie Mellon University
Pittsburgh, PA, USA
srchvrs@cmu.edu

Bilegsaikhan Naidan
Norwegian University of Science and Technology
Trondheim, Norway
bileg@idi.ntnu.no

Abstract

Our focus is on approximate nearest neighbor retrieval in metric and non-metric spaces. We employ a VP-tree and explore two simple yet effective learning-to-prune approaches: density estimation through sampling and "stretching" of the triangle inequality. Both methods are evaluated using data sets with metric (Euclidean) and non-metric (KL-divergence and Itakura-Saito) distance functions. Conditions on spaces where the VP-tree is applicable are discussed. The VP-tree with a learned pruner is compared against the recently proposed state-of-the-art approaches: the bbtree, the multi-probe locality sensitive hashing (LSH), and permutation methods. Our method was competitive against state-of-the-art methods and, in most cases, was more efficient for the same rank approximation quality.

1 Introduction

Similarity search algorithms are essential to multimedia retrieval, computational biology, and statistical machine learning. Resemblance between objects x and y is typically expressed in the form of a distance function $d(x, y)$, where smaller values indicate less dissimilarity. In our work we use the Euclidean distance (L_2), the KL-divergence ($\sum x_i \log x_i/y_i$), and the Itakura-Saito distance ($\sum x_i/y_i - \log x_i/y_i - 1$). KL-divergence is commonly used in text analysis, image classification, and machine learning [6]. Both KL-divergence and the Itakura-Saito distance belong to a class of distances called Bregman divergences.

Our interest is in the nearest neighbor (NN) search, i.e., we aim to retrieve the object o that is closest to the query q. For the KL-divergence and other non-symmetric distances two types of NN-queries are defined. The *left* NN-query returns the object o that minimizes the distance $d(o, q)$, while the *right* NN-query finds o that minimizes $d(q, o)$.

The distance function can be computationally expensive. There was a considerable effort to reduce computational costs through approximating the distance function, projecting data in a low-dimensional space, and/or applying a hierarchical space decomposition. In the case of the hierarchical space decomposition, a retrieval process is a recursion that employs an "oracle" procedure. At each step of the recursion, retrieval can continue in one or more partitions. The oracle allows one to prune partitions without directly comparing the query against data points in these partitions. To this end, the oracle assesses the query and estimates which partitions may contain an answer and, therefore, should be recursively analyzed. A pruning algorithm is essentially a binary classifier. In metric spaces, one can use the classifier based on the triangle inequality. In non-metric spaces, a classifier can be learned from data.

There are numerous data structures that speedup the NN-search by creating hierarchies of partitions at index time, most notably the VP-tree [28, 31] and the KD-tree [4]. A comprehensive review of these approaches can be found in books by Zezula et al. [32] and Samet [27]. As dimensionality

increases, the filtering efficiency of space-partitioning methods decreases rapidly, which is known as the "curse of dimensionality" [30]. This happens because in high-dimensional spaces histograms of distances and 1-Lipschitz function values become concentrated [25]. The negative effect can be partially offset by creating overlapping partitions (see, e.g., [21]) and, thus, trading index size for retrieval time. The approximate NN-queries are less affected by the curse of the dimensionality, because it is possible to reduce retrieval time at the cost of missing some relevant answers [18, 9, 25]. Low-dimensional data sets embedded into a high-dimensional space do not exhibit high concentration of distances, i.e., their *intrinsic* dimensionality is low. In metric spaces, it was proposed to compute the intrinsic dimensionality as the half of the squared signal to noise ratio (for the distance distribution) [10].

A well-known approximate NN-search method is the locality sensitive hashing (LSH) [18, 17]. It is based on the idea of random projections [18, 20]. There is also an extension of the LSH for *symmetric* non-metric distances [23]. The LSH employs several hash functions: It is likely that close objects have same hash values and distant objects have different hash values. In the classic LSH index, the probability of finding an element in one hash table is small and, consequently, many hash tables are to be created during indexing. To reduce space requirements, Lv et al. proposed a multi-probe version of the LSH, which can query multiple buckets of the same hash table [22]. Performance of the LSH depends on the choice of parameters, which can be tuned to fit the distribution of data [11].

For approximate searching it was demonstrated that an early termination strategy could rely on information about distances from typical queries to their respective nearest neighbors [33, 1]. Amato et al. [1] showed that density estimates can be used to approximate a pruning function in metric spaces. They relied on a hierarchical decomposition method (an M-tree) and proposed to visit partitions in the order defined by density estimates. Chávez and Navarro [9] proposed to relax triangle-inequality based lower bounds for distances to potential nearest neighbors. The approach, which they dubbed as *stretching of the triangle inequality*, involves multiplying an exact bound by $\alpha > 1$.

Few methods were designed to work in non-metric spaces. One common indexing approach involves mapping the data to a low-dimensional Euclidean space. The goal is to find the mapping without large distortions of the original similarity measure [19, 16]. Jacobs et al. [19] review various projection methods and argue that such a coercion is often against the nature of a similarity measure, which can be, e.g., intrinsically non-symmetric. A mapping can be found using machine learning methods. This can be done either separately for each data point [12, 24] or by computing one global model [3]. There are also a number of approaches, where machine learning is used to estimate optimal parameters of classic search methods [7]. Vermorel [29] applied VP-trees to searching in undisclosed non-metric spaces without trying to learn a pruning function. Like Amato et al. [1], he proposed to visit partitions in the order defined by density estimates and employed the same early termination method as Zezula et al. [33].

Cayton [6] proposed a Bregman ball tree (bbtree), which is an exact search method for Bregman divergences. The bbtree divides data into two clusters (each covered by a Bregman ball) and recursively repeats this procedure for each cluster until the number of data points in a cluster falls below a threshold (a bucket size). At search time, the method relies on properties of Bregman divergences to compute the shortest distances to covering balls. This is an expensive iterative procedure that may require several computations of direct and inverse gradients, as well as of several distances. Additionally, Cayton [6] employed an early termination method: The algorithm can be told to stop after processing a pre-specified number of buckets. The resulting method is an approximate search procedure. Zhang et al. [34] proposed an exact search method based on estimating the maximum distance to a bounding rectangle, but it works with *left queries* only. The most efficient variant of this method relies on an optimization technique applicable only to certain decomposable Bregman divergences (a decomposable distance is a sum of values computed separately for each coordinate).

Chávez et al. [8] as well as Amato and Savino [2] independently proposed permutation-based search methods. These approximate methods do not involve learning, but, nevertheless, are applicable to non-metric spaces. At index time, k pivots are selected. For every data point, we create a list, called a *permutation*, where pivots are sorted in the order of increasing distances from the data point. At query time, a rank correlation (e.g., Spearman's) is computed between the permutation of the query and permutations of data points. Candidate points, which have sufficiently small correlation values, are then compared directly with the query (by computing the original distance function). One can sequentially scan the list of permutations and compute the rank correlation between the

permutation of the query and the permutation of every data point [8]. Data points are then sorted by rank-correlation values. This approach can be improved by incremental sorting [14], storing permutations as inverted files [2], or prefix trees [13].

In this work we experiment with two approaches to learning a pruning function of the VP-tree, which to our knowledge was not attempted previously. We compare the resulting method, which can be applied to both metric and non-metric spaces, with the following state-of-the-art methods: the multi-probe LSH, permutation methods, and the bbtree.

2 Proposed Method

2.1 Classic VP-tree

In the VP-tree (also known as a ball tree) the space is partitioned with respect to a (usually randomly) chosen pivot π [28, 31]. Assume that we have computed distances from all points to the pivot π and R is a median of these distances. The sphere centered at π with the radius R divides the space into two partitions, each of which contains approximately half of all points. Points inside the pivot-centered sphere are placed into the left subtree, while points outside the pivot-centered sphere are placed into the right subtree (points on the border may be placed arbitrarily). The search algorithm proceeds recursively. When the number of data points is below a certain threshold (the bucket size), these data points are stored as a single bucket. The obtained hierarchical partition is represented by the binary tree, where buckets are leaves.

The NN-search is a recursive traversal procedure that starts from the root of the tree and iteratively updates the distance r to the closest object found. When it reaches a bucket (i.e., a leaf), bucket elements are searched sequentially. Each internal node stores the pivot π and the radius R. In a metric space with the distance $d(x, y)$, we use the triangle inequality to prune the search space. We visit:

- *only* the left subtree if $d(\pi, q) < R - r$;
- *only* the right subtree if $d(\pi, q) > R + r$;
- both subtrees if $R - r \leq d(\pi, q) \leq R + r$.

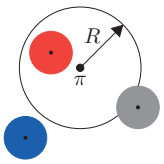


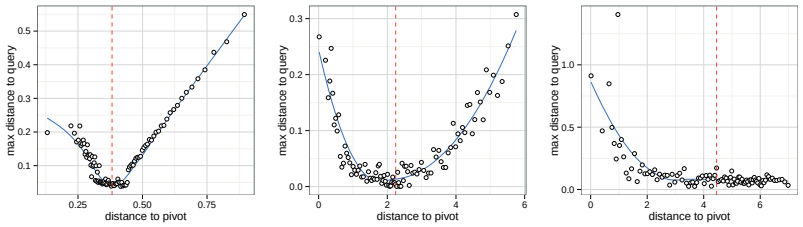
Figure 1: Three types of query balls in the VP-tree. The black circle (centered at the pivot π) is the sphere that divides the space.

In the third case, we first visit the partition that contains q. These three cases are illustrated in Fig. 1. Let $D_{\pi,R}(x) = |R - x|$. Then we need to visit both partitions if and only if $r \geq D_{\pi,R}(d(\pi, q))$. If $r < D_{\pi,R}(d(\pi, q))$, we visit only the partition containing the query point. In this case, we prune the other partition. Pruning is a *classification task* with three classes, where the prediction function is defined through $D_{\pi,R}(x)$. The only argument of this function is a distance between the pivot and the query, i.e., $d(\pi, q)$. The function value is equal to the maximum radius of the query ball that fits inside the partition containing the query (see the red and the blue sample balls in Fig. 1).

2.2 Approximating $D_{\pi,R}(x)$ with a Piece-wise Linear Function

In Section 2 of the supplemental materials, we describe a straightforward sampling algorithm to learn the decision function $D_{\pi,R}(x)$ for every pivot π. This method turned out to be inferior to most state-of-the-art approaches. It is, nevertheless, instructive to examine the decision functions $D_{\pi,R}(x)$ learned by sampling for the Euclidean distance and KL-divergence (see Table 1 for details on data sets).

Each point in Fig. 2a-2c is a value of the decision function obtained by sampling. Blue curves are fit to these points. For the Euclidean data (Fig. 2a), $D_{\pi,R}(x)$ resembles a piece-wise linear function approximately equal to $|R - x|$. For the KL-divergence data (Fig. 2b and 2c), $D_{\pi,R}(x)$ looks like a U-shape and a hockey-stick curve, respectively. Yet, most data points concentrate around the median (denoted by a dashed red line). In this area, a piece-wise linear approximation of $D_{\pi,R}(x)$ could



(a) Colors, L_2 (b) RCV-8, KL-divergence (c) RCV-16, gen. KL-divergence

Figure 2: The empirically obtained decision function $D_{\pi,R}(x)$. Each point is a value of the function learned by sampling (see Section 2 of the supplemental materials). Blue curves are fit to these points. The red dashed line denotes a median distance R from data set points to the pivot π.

still be reasonable. Formally, we define the decision function as:

$$D_{\pi,R}(x) = \begin{cases} \alpha_{left}|x - R|, & \text{if } x \leq R \\ \alpha_{right}|x - R|, & \text{if } x \geq R \end{cases} \tag{1}$$

Once we obtain the values of α_{left} and α_{right} that permit near exact searching, we can induce more aggressive pruning by increasing α_{left} and/or α_{right}, thus, exploring trade-offs between retrieval efficiency and effectiveness. This is similar to stretching of the triangle inequality proposed by Chávez and Navarro [9].

Optimal α_{left} and α_{right} are determined using a grid search. To this end, we index a small subset of the data points and seek to obtain parameters that give the shortest retrieval time at a specified recall threshold. The grid search is initialized by values a and b. Then, recall values and retrieval times for all $\alpha_{left} = a\rho^{i/m\ \ 0.5}$ and $\alpha_{right} = b\rho^{j/m\ \ 0.5}$ are obtained ($1 \leq i, j \leq m$). The values of m and ρ are chosen so that: (1) the grid step is reasonably small (i.e., $\rho^{1/m}$ is close to one); (2) the search space is manageable (i.e., m is not large).

If the obtained recall values are considerably larger than a specified threshold, the procedure repeats the grid search using larger values of a and b. Similarly, if the recall is not sufficient, the values of a and b are decreased and the grid search is repeated. One can see that the perfect recall can be achieved with $\alpha_{left} = 0$ and $\alpha_{right} = 0$. In this case, no pruning is done and the data set is searched sequentially. Values of $\alpha_{left} = \infty$ and $\alpha_{right} = \infty$ represent an (almost) zero recall, because one of the partitions is always pruned.

2.3 Applicability Conditions

It is possible to apply the classic VP-tree algorithm only to data sets such that $D_{\pi,R}(d(\pi, q)) > 0$ when $d(\pi, q) \neq R$. In a relaxed version of this applicability condition, we require that $D_{\pi,R}(d(\pi, q)) > 0$ for almost all queries and a large subset of data points. More formally:

Property 1. *For any pivot π, probability α, and distance $x \neq R$, there exists a radius $r > 0$ such that, if two randomly selected points q (a potential query) and u (a potential nearest neighbor) satisfy $d(\pi, q) = x$ and $d(u, q) \leq r$, then both p and q belong to the same partition (defined by π and R) with a probability at least α.*

The Property 1, which is true for all metric spaces due to the triangle inequality, holds in the case of the KL-divergence and data points u sampled randomly and uniformly from the simplex $\{x_i | x_i \geq 0, \sum x_i = 1\}$. The proof, which is given in Section 1 of supplemental materials, can be trivially extended to other non-negative distance functions $d(x, y) \geq 0$ (e.g., to the Itakura-Saito distance) that satisfy (additional compactness requirements may be required): (1) $d(x, y) = 0 \Leftrightarrow x = y$; (2) the set of discontinuities of $d(x, y)$ has measure zero in L_2. This suggests that the VP-tree could be applicable to a wide class of non-metric spaces.

Table 1: Description of the data sets

Name	$d(x, y)$	Data set size	Dimensionality	Source
Colors	L_2	$1.1 \cdot 10^5$	112	Metric Space Library[1]
RCV-i	KL-div, L_2	$0.5 \cdot 10^6$	$i \in \{8, 16, 32, 128, 256\}$	Cayton [6]
SIFT-signat.	KL-div, L_2	$1 \cdot 10^4$	1111	Cayton [6]
Uniform	L_2	$0.5 \cdot 10^6$	64	Sampled from $U^{64}[0, 1]$

3 Experiments

We run experiments on a Linux server equipped with Intel Core i7 2600 (3.40 GHz, 8192 KB of L3 CPU cache) and 16 GB of DDR3 RAM (transfer rate is 20GB/sec). The software (including scripts that can be used to reproduce our results) is available online, as a part of the *Non-Metric Space Library*[2] [5]. The code was written in C++, compiled using GNU C++ 4.7 (optimization flag -Ofast), and executed in a single thread. SIMD instructions were enabled using the flags -msse2 -msse4.1 -mssse3.

All distance and rank correlation functions are highly optimized and employ SIMD instructions. Vector elements were single-precision numbers. For the KL-divergence, though, we also implemented a slower version, which computes logarithms on-line, i.e., for each distance computation. The faster version computes logarithms of vector elements off-line, i.e., during indexing, and stores with the vectors. Additionally, we need to compute logarithms of query vector elements, but this is done only once per query. The optimized implementation is about 30x times faster than the slower one.

The data sets are described in Table 1. Each data set is randomly divided into two parts. The smaller part (containing 1,000 elements) is used as queries, while the larger part is indexed. This procedure is repeated 5 times (for each data sets) and results are aggregated using a classic fixed-effect model [15]. *Improvement in efficiency* due to indexing is measured as a reduction in retrieval time compared to a sequential, i.e., exhaustive, search. The effectiveness was measured using a simple rank error metric proposed by Cayton [6]. It is equal to the number of NN-points closer to the query than the nearest point returned by the search method. This metric is appropriate mostly for 1-NN queries. We present results only for left queries, but we also verified that in the case of right queries the VP-tree provides similar effectiveness/efficiency trade-offs. We ran benchmarks for L_2, the KL-divergence,[3] and the Itakura-Saito distance. Implemented methods included:

- The novel search algorithm based on the VP-tree and a piece-wise linear approximation for $D_{\pi,R}(x)$ as described in Section 2.2. The parameters of the grid search algorithm were: $m = 7$ and $\rho = 8$.
- The permutation method with incremental sorting [14]. The near-optimal performance was obtained by using 16 pivots.
- The permutation prefix index, where permutation profiles are stored in a prefix tree of limited depth [13]. We used 16 pivots and the maximal prefix length 4 (again selected for best performance).
- The bbtree [6], which is designed for Bregman divergences, and, thus, it was not used with L_2.
- The multi-probe LSH, which is designed to work only for L_2. The implementation employs the LSHKit, [4] which is embedded in the *Non-Metric Space Library*. The best-performing configuration that we could find used 10 probes and 50 hash tables. The remaining parameters were selected automatically using the cost model proposed by Dong et al. [11].

[2] `https://github.com/searchivarius/NonMetricSpaceLib`

[3] In the case of SIFT signatures, we use generalized KL-divergence (similarly to Cayton).

[4] Downloaded from `http://lshkit.sourceforge.net/`

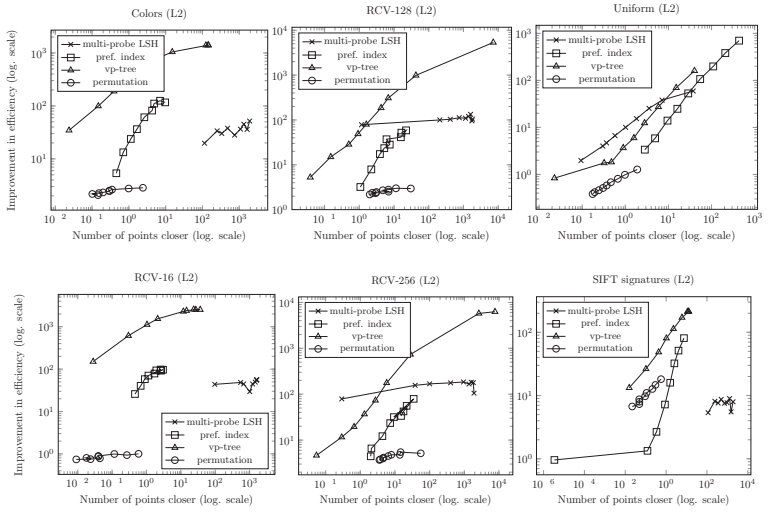


Figure 3: Performance of NN-search for L_2

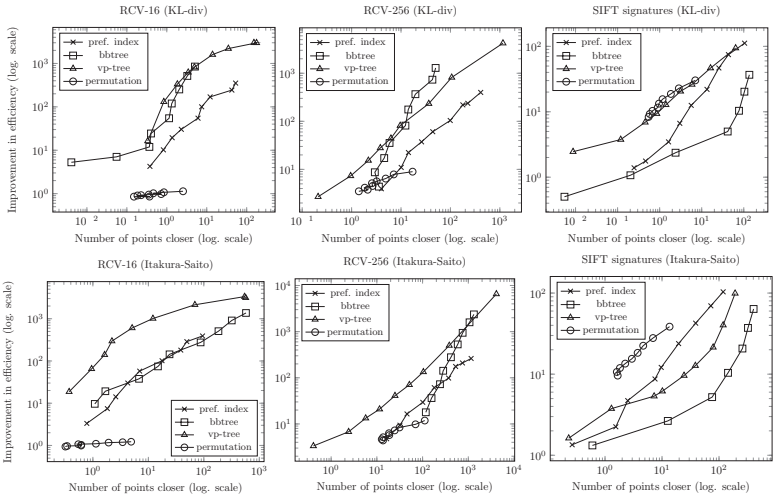


Figure 4: Performance of NN-search for the KL-divergence and Itakura-Saito distance

For the bbtree and the VP-tree, vectors in the same bucket were stored in contiguous chunks of memory (allowing for about 1.5-2x reduction in retrieval times). It is typically more efficient to search elements of a small bucket sequentially, rather than using an index. A near-optimal performance was obtained with 50 elements in a bucket. The same optimization approach was also used for both permutation methods.

Several parameters were manually selected to achieve various effectiveness/efficiency trade-offs. They included: the minimal number/percentage of candidates in permutation methods, the desired

Table 2: Improvement in efficiency and retrieval time (ms) for the bbtree without early termination

Data set	RCV-16		RCV-32		RCV-128		RCV-256		SIFT sign.	
	impr.	time	impr.	time	impr.	time	impr.	time	impr.	time
Slow KL-divergence	15.7	8	6.7	36	1.6	613	1.1	1700	0.9	164
Fast KL-divergence	4.6	2.5	1.9	9.6	0.5	108	0.4	274	0.4	18

recall in the multi-probe LSH and in the VP-tree, as well as the maximum number of processed buckets in the bbtree.

The results for L_2 are given in Fig. 3. Even though a representational dimensionality of the Uniform data set is only 64, it has the highest intrinsic dimensionality among all sets in Table 1 (according to the definition of Chávez et al. [10]). Thus, for the Uniform data set, no method achieved more than a 10x speedup over sequential searching without substantial quality degradation. For instance, for the VP-tree, a 160x speedup was only possible, when a retrieved object was a 40-th nearest neighbor (on average) instead of the first one. The multi-probe LSH can be twice as fast as the VP-tree at the expense of having a 4.7x larger index. All the remaining data sets have low or moderate intrinsic dimensionality (smaller than eight). For example, the SIFT signatures have the representational dimensionality of 1111, but the intrinsic dimensionality is only four. For data sets with low and moderate intrinsic dimensionality, the VP-tree is faster than the other methods most of the time. For the data sets Colors and RCV-16 there is a two orders of magnitude difference.

The results for the KL-divergence and Itakura-Saito distance are summarized in Fig. 4. The bbtree is never substantially faster than the VP-tree, while being up to an order of magnitude slower for RCV-16 and RCV-256 in the case of Itakura-Saito distance. Similar to results in L_2, in most cases, the VP-tree is at least as fast as other methods. Yet, for the SIFT signatures data set and the Itakura-Saito distance, permutation methods can be twice as fast.

Additional analysis has showed that the VP-tree is a good rank-approximation method, but it is not necessarily the best approach in terms of recall. When the VP-tree misses the nearest neighbor, it often returns the second nearest or the third nearest neighbor instead. However, when other examined methods miss the nearest neighbor, they frequently return elements that are far from the true result. For example, the multi-probe LSH may return a true nearest neighbor 50% of the time, and 50% of the time it would return the 100-th nearest neighbor. This observation about the LSH is in line with previous findings [26].

Finally, we measured improvement in efficiency (over exhaustive search) for the bbtree, where the early termination algorithm was disabled. This was done using both the slow and the fast implementation of the KL-divergence. The results are given in Table 2. Improvements in efficiency for the case of the slower KL-divergence (reported in the first row) are consistent with those reported by Cayton [6]. The second row shows improvements in efficiency for the case of the faster KL-divergence and these improvements are substantially smaller than those reported in the first row, despite the fact that using the faster KL-divergence greatly reduces retrieval times. The reason is that the pruning algorithm of the bbtree is quite expensive. It involves computations of logarithms/exponents for coordinates of unknown vectors, and, thus, these computations cannot be deferred to index time.

4 Discussion and conclusions

We evaluated two simple yet effective learning-to-prune methods and showed that the resulting approach was competitive against state-of-the-art methods in both metric and non-metric spaces. In most cases, this method provided better trade-offs between rank approximation quality and retrieval speed. For datasets with low or moderate intrinsic dimensionality, the VP-tree could be one-two orders of magnitude faster than other methods (for the same rank approximation quality). We discussed applicability of our method (a VP-tree with the learned pruner) and proved a theorem supporting the point of view that our method can be applicable to a class of non-metric distances, which includes

the KL-divergence. We also showed that a simple trick of pre-computing logarithms at index time substantially improved performance of existing methods (e.g., bbtree) for the studied distances.

It should be possible to improve over basic learning-to-prune methods (employed in this work) using: (1) a better pivot-selection strategy [31]; (2) a more sophisticated sampling strategy; (3) a more accurate (non-linear) approximation for the decision function $D_{\pi,R}(x)$ (see section 2.1).

5 Acknowledgements

We thank Lawrence Cayton for providing the data sets, the bbtree code, and answering our questions; Anna Belova for checking the proof of Property 1 (supplemental materials) and editing the paper.

References

[1] G. Amato, F. Rabitti, P. Savino, and P. Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2):192–227, Apr. 2003.

[2] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd international conference on Scalable information systems*, InfoScale '08, pages 28:1–28:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: A method for efficient approximate similarity rankings. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–268 – II–275 Vol.2, june-2 july 2004.

[4] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] L. Boytsov and B. Naidan. Engineering efficient and effective Non-Metric Space Library. In N. Brisaboa, O. Pedreira, and P. Zezula, editors, *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer Berlin Heidelberg, 2013.

[6] L. Cayton. Fast nearest neighbor retrieval for Bregman divergences. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 112–119, New York, NY, USA, 2008. ACM.

[7] L. Cayton and S. Dasgupta. A learning framework for nearest neighbor search. *Advances in Neural Information Processing Systems*, 20, 2007.

[8] E. Chávez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(9):1647 –1658, sept. 2008.

[9] E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85(1):39–46, 2003.

[10] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[11] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 669–678, New York, NY, USA, 2008. ACM.

[12] O. Edsberg and M. L. Hetland. Indexing inexact proximity search with distance regression in pivot space. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, pages 51–58, New York, NY, USA, 2010. ACM.

[13] A. Esuli. Use of permutation prefixes for efficient and scalable approximate similarity search. *Inf. Process. Manage.*, 48(5):889–902, Sept. 2012.

[14] E. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(9):1647–1658, 2008.

[15] L. V. Hedges and J. L. Vevea. Fixed-and random-effects models in meta-analysis. *Psychological methods*, 3(4):486–504, 1998.

[16] G. Hjaltason and H. Samet. Properties of embedding methods for similarity searching in metric spaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(5):530–549, 2003.

[17] P. Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of discrete and computational geometry*, pages 877–892. Chapman and Hall/CRC, 2004.

[18] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[19] D. Jacobs, D. Weinshall, and Y. Gdalyahu. Classification with nonmetric distances: Image retrieval and class representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(6):583–600, 2000.

[20] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th annual ACM symposium on Theory of computing*, STOC '98, pages 614–623, New York, NY, USA, 1998. ACM.

[21] H. Lejsek, F. Ásmundsson, B. Jónsson, and L. Amsaleg. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(5):869 –883, may 2009.

[22] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[23] Y. Mu and S. Yan. Non-metric locality-sensitive hashing. In *AAAI*, 2010.

[24] T. Murakami, K. Takahashi, S. Serita, and Y. Fujii. Versatile probability-based indexing for approximate similarity search. In *Proceedings of the Fourth International Conference on SImilarity Search and APplications*, SISAP '11, pages 51–58, New York, NY, USA, 2011. ACM.

[25] V. Pestov. Indexability, concentration, and $\{\mathrm{VC}\}$ theory. *Journal of Discrete Algorithms*, 13(0):2 – 18, 2012. Best Papers from the 3rd International Conference on Similarity Search and Applications (SISAP 2010).

[26] P. Ram, D. Lee, H. Ouyang, and A. G. Gray. Rank-approximate nearest neighbor search: Retaining meaning and speed in high dimensions. In *Advances in Neural Information Processing Systems*, pages 1536–1544, 2009.

[27] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers Inc., 2005.

[28] J. Uhlmann. Satisfying general proximity similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[29] J. Vermorel. Near neighbor search in metric and nonmetric space, 2005. `http://hal.archives-ouvertes.fr/docs/00/03/04/85/PDF/densitree.pdf` last accessed on Nov 1st 2012.

[30] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 194–205. Morgan Kaufmann, August 1998.

[31] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[32] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[33] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with M-trees. *The VLDB Journal*, 7(4):275–293, Dec. 1998.

[34] Z. Zhang, B. C. Ooi, S. Parthasarathy, and A. K. H. Tung. Similarity search on Bregman divergence: towards non-metric indexing. *Proc. VLDB Endow.*, 2(1):13–24, Aug. 2009.

Learning to Prune in Metric and Non-Metric Spaces (supplemental material)

Leonid Boytsov
Carnegie Mellon University
Pittsburgh, PA, USA
srchvrs@cmu.edu

Bilegsaikhan Naidan
Norwegian University of Science and Technology
Trondheim, Norway
bileg@idi.ntnu.no

1 Introduction

This short note supplements the paper "Learning to Prune in Metric and Non-Metric Spaces" [3]. We aim to provide a theoretical justification for applicability of our approach (the VP-tree with a learned pruner) to a class of non-metric spaces, which includes the KL-divergence and the Itakura-Saito distance. In addition, we describe a simple algorithm to learn the decision function through sampling.

2 Applicability Conditions

Theorem 1. *For any pivot π, probability α, and distance $x \neq R$, there exists a radius $r > 0$ such that, if two randomly selected points q (a potential query) and u (a potential nearest neighbor) satisfy $d(\pi, q) = x$ and $d(u, q) \leq r$, then both p and q belong to the same partition (defined by π and R) with a probability at least α.*

Theorem 1, which is true for all metric spaces, holds in the case of KL-divergence and data points u sampled randomly and uniformly from the simplex $\{x_i | x_i \geq 0, \sum x_i = 1\}$.

Note 1. *This theorem is trivially extended to many other* non-negative *distance functions $d(x, y)$ that satisfy:*

- *$d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$;*
- *$d(x, y)$ is continuous except for a set of measure zero;*

In particular, the theorem holds for the Itakura-Saito distance. Note, though, that these conditions are not *sufficient*, because one may need to make additional compactness requirements. For the example, the proof for the KL-divergence relies on the fact that the distance is defined on the compact Euclidean subset.

Proof. It is easy to show that for any α there exists $\epsilon > 0$ such that all coordinates of the randomly selected vector are $\geq \epsilon$ with a probability at least α. Further, we consider the compact set of vectors (it is compact with respect to L_2):

$$S(\epsilon) = \{y | 1 \geq y_i \geq \epsilon, \sum y_i = 1\}$$

The KL-divergence is defined as:

$$KL(x, y) = \sum_i x_i \log \frac{x_i}{y_i}$$

For any $y \in S(\epsilon)$, $y_i \geq \epsilon$. Thus, $KL(x, y)$ is a continuous function of both arguments on $S \times S$. Points outside S are encountered with probability $1 - \alpha$, which can be made arbitrarily small by selecting a sufficiently small ϵ.

For the sake of contradiction, we assume that, no matter how small is the query radius, there is a query ball with the center in S at distance r from the pivot. In addition, there are points inside query balls that belong to both partitions as well as to S. This can be seen as an adversarial game, where we select progressively decreasing radii $r_n \to 0$. For each r_n our adversary finds the query ball with the center $q_n \in S$ and the radius $\leq r_n$ such that (1) $KL(\pi, q_n) = x$ and (2) the query ball intersects both space partitions and S. To demonstrate the latter, our adversary provides us with points u_n^+ and u that lie inside the query ball and belong to different space partitions. Note that q_n, u_n^+, and u should all belong to S.

Formally, there exists a sequence of radii $r_n \to 0$, the sequence of query ball centers q_n, and sequences of points u_n^+, u_n such that:

$$KL(\pi, q_n) = x,$$

$$KL(u_n^+, q_n) \leq r_n \text{ and } KL(u_n, q_n) \leq r_n,$$

but

$$KL(\pi, u_n) < R \text{ and } KL(\pi, u_n^+) > R. \tag{1}$$

The sequence (q_n, u_n^+, u_n) is defined on a Cartesian product $S \times S \times S$, which is compact due to Tychonoff's theorem. Because the Cartesian product is compact, we can assume that (q_n, u_n, u_n^+) is a converging sequence and sequences q_n, u_n, u_n^+ converge as well: [1]:

$$(q_n, u_n, u_n^+) \to (q, u, u^+) \tag{2}$$

From Eq. 1-2 and continuity of the function $KL(x, y)$ on $S \times S$, we obtain:

$$KL(q, u^+) = KL(q, u) = 0, \tag{3}$$

$$KL(\pi, u^+) \geq R, \; KL(\pi, u) \leq R. \tag{4}$$

From properties of the KL-divergence and Eq. 2, it follows that $u^+ = q = u$. By applying $u^+ = q = u$ to Eq. 4, we get that $R \leq KL(\pi, q) \leq R$ and, thus, that:

$$KL(\pi, q) = R. \tag{5}$$

Again, from continuity of $KL(x, y)$, $KL(\pi, q_n) = x$ and $q_n \to q$, we obtain that $KL(\pi, q) = x$. Because $x \neq R$ this conclusion contradicts to Eq. 5.

We obtained a contradiction, which demonstrates that (almost) all sufficiently small query balls at distance $r \neq R$ from the pivot lie (for the most part) in either the left or the right partition. The exceptions are query balls centered outside S, or query ball parts that don't belong to S. Yet, as noted previously, it is possible to select S such that a probability of drawing a point from S will be arbitrarily close to 1. This observation finishes the proof of the theorem. □

3 Estimating Decision Function $D_{\pi,R}(x)$ Through Sampling

It is possible to estimate $D_{\pi,R}(x)$ (defined in Section 2.2 of [3]) through sampling. Note that the resulting search method would not be exact. A straightforward sampling method involves random and independent selection of points q_i and u_i from the data set. Two cases are possible depending on whether q_i and u_i belong to the same partition. Consider the case when q_i and u_i belong to different partitions. This represents the gray ball from Fig. 1. Thus, we learn that there may exist multiple pairs of points (different from q_i) within the distance $r = d(u_i, q_i)$ from q_i situated in different partitions. Thus, we can be absolutely sure that $D_{\pi,R}(d(\pi, q_i)) \leq d(u_i, q_i)$.

[1] If a space X is compact any sequence contains a converging subsequence with the limit in X.

In the case when u_i is in the same partitions as q_i, we cannot, however, infer that $D_{\pi,R}(d(\pi, q_i)) > d(u_i, q_i)$. Indeed, there could exist a nearest-neighbor u_j, not encountered by the sampling procedure, belonging to a *different* space partition than q_i, but, nevertheless, satisfying: $d(u_j, q_j) \leq d(u_i, q_i)$. If we use q_i as a query and set $D_{\pi,R}(d(\pi, q_i))$ to be larger than $d(u_i, q_i)$, the partition containing u_j will be pruned and, consequently, u_j will not be found.

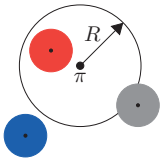


Figure 1: Three types of query balls in the VP-tree. The black circle (centered at the pivot π) is the sphere that divides the space.

By repeating the sampling procedure multiple times and smoothing results (e.g., by fitting a curve or learning a regression model), we can obtain an estimate for the upper bound of $D_{\pi,R}(x)$. There are several problems with this approach. First, due to the concentration of measure, $d(\pi, q_i)$ is close to R for most sampled points. Thus, $D_{\pi,R}(x)$ will be properly estimated only for values $x \approx R$. Second, it does not allow us to trade search effectiveness for efficiency.

The underlying principle of an improved sampling method is to divide the xy-plane, which represents the plot of the function $D_{\pi,R}(x)$, into cells. This improved sampling method works as follows:

- We compile the distribution of distances $d(\pi, q_i)$ (using all data points) and divide it into 50-500 quantiles. These "horizontal" quantiles represent the division of the xy-plane into vertical bars. Then, several pseudo-queries q_i are randomly picked from each horizontal quantile.
- For each pseudo-query q_i, $K \approx 100$ pseudo near-neighbors are randomly selected from the data set. We also implemented an approach where K *true* near-neighbors are obtained by exhaustively searching the data set (an idea proposed by Athitsos et al. [2]). This method is computationally expensive, but it did not result in substantial improvements.
- Now each vertical bar contains a number of pseudo near neighbors. We compute the bar-specific distributions of distances from q_i to these points and divide each of the distributions into 100-1000 "vertical" quantiles. This step finalizes a division of the xy-plane into rectangular cells.

To estimate $D_{\pi,R}(x)$, we find the vertical bar containing the point $(x, 0)$. Then, we start scanning the cells belonging to this bar in the bottom-up fashion. The algorithm keeps two counters. The first counter N_{all} is a total number of of pseudo near neighbors contained in the visited cells. The second counter N_{diff} is the number neighbors that belong to a different partition than their respective pseudo queries (i.e., the number of situations when we have the gray pseudo query ball, see Fig. 1.). We stop when N_{diff} becomes larger than γN_{all} for some threshold value γ. A y-coordinate corresponding to the last visited cell is used as an estimate for $D_{\pi,R}(x)$: One can use the minimum, the maximum, or any intermediate y-coordinate of points inside the last visited cell. The threshold γ is selected empirically. In that, highest recall values (and slowest speeds) are obtained for $\gamma = 0$. Unlike previous efforts, see e.g. [1], our sampling algorithm estimates $D_{\pi,R}(x)$ for every pivot π (rather than one global distribution) and, thus, it may better adapt to specifics of data partitions induced by pivots.

References

[1] G. Amato, F. Rabitti, P. Savino, and P. Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2):192–227, Apr. 2003.

[2] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: A method for efficient approximate similarity rankings. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–268 – II–275 Vol.2, june-2 july 2004.

[3] L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *Advances in Neural Information Processing Systems*, 2013.

Paper V: Shrinking data balls in metric indexes

Bilegsaikhan Naidan and Magnus Lie Hetland.
Appeared at The 5th International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA), 2013.

Appendix E

Shrinking data balls in metric indexes

Bilegsaikhan Naidan and Magnus Lie Hetland
Department of Computer and Information Science,
Norwegian University of Science and Technology,
Sem Sælands vei 7-9, NO-7491 Trondheim, Norway
{*bileg,mlh*}*@idi.ntnu.no*

***Abstract*—Some of the existing techniques for approximate similarity retrieval in metric spaces are focused on shrinking the query region by user-defined parameter. We modify this approach slightly and present a new approximation technique that shrinks data regions instead. The proposed technique can be applied to any metric indexing structure based on the ball-partitioning principle. Experiments show that our technique performs better than the relative error approximation and region proximity techniques, and that it achieves significant speedup over exact search with a low degree of error. Beyond introducing this new method, we also point out and remedy a problem in the relative error approximation technique, substantially improving its performance.**

***Keywords*-approximation algorithms, experiments, similarity search, metric space.**

I. Introduction

Nowadays, efficient similarity retrieval is becoming more important in various applications such as multimedia repositories (images, audio, video) because of the rapid growth of these data sets and the increasing demand for access to them. In such search applications, the relevance of a data object is often measured by some distance function that provides quantitative information about its similarity to some given sample query. For search techniques that treat the distance as a black-box relevance measure, the main challenge is to quickly retrieve a small set of the most relevant objects (either all within a search radius, or the k nearest neighbors, k-NN) relying on the properties of the distance—usually by exploiting the metric axioms.

Numerous metric indexing structures have been proposed to reduce the computational cost (such as the total number of distance computations at query time) of similarity retrieval [1]. These methods primarily rely on various forms of filtering based on the triangle inequality. Triangular filtering is efficient in low-dimensional spaces. However, as the dimensionality of a space increases, the performance of these indexes degrades because of the so-called curse of dimensionality: distances grow increasingly similar, and eventually one may need to examine more or less all data objects, the equivalent of a linear scan. One promising approach to ameliorating this curse is *approximate* similarity search, where some result quality is sacrificed in order to gain performance. This is acceptable in many applications, as distance-based retrieval is generally approximate to begin with—the distance function is most likely an approximation of the user's perception of similarity, and the user probably wants *similar* objects (e.g., pictures of horses), not necessarily the *most* similar objects (i.e., the most similar horse).

Some important methods used in approximate similarity search are discarding data regions at query time (by shrinking the query ball by a user-defined factor [2–4] or by analyzing the intersection of query and data regions [5]), representing data objects as permutations of a set of pivots [6], and estimating the distance by linear regression [7]. We focus on the first approach, trying to develop a method for discard regions that overlap with the query, but that are likely to contain few relevant objects, if any. Our main contributions are:

- We propose a new approximation technique that shrinks data regions instead of the query region, and show empirically that it is superior to existing methods in many cases.
- We amend a problem in the relative error approximation technique of Zezula et al. [2]. In several experimental studies, this technique was found to be the worst one [1, 2, 5]. We point out a problem with how the method has been used, and show how the amended version has significantly improved performance wrt. the original, making it comparable even to the region proximity method [5].

The rest of the paper is organized as follows. In Section II, we briefly review related work. In Section III, we propose our approximation technique, and describe how to amend the relative error approximation technique. Section IV provides experimental results and some discussion of those results. Finally, Section V contains some concluding remarks.

II. Related Work

In this section, we briefly review two metric indexing structures based on the ball-partitioning principle—the M- and SSS-trees—explain some issues in the construction of indexes and also review some approximate techniques that can be applied to those structures.

The M-tree [8] is a hierarchical dynamic metric ball tree that is designed for secondary memory. The M-tree is built in a bottom-up manner like B-trees. The insertion algorithm starts from the root and moves toward the leaves by selecting

nodes that are closer to the new object or that require a minimum enlargement of existing balls. The new object is finally inserted into a leaf node. This may cause the leaf to split (if the node capacity is exceeded), which may trigger splits in some of its ancestor nodes, possibly even the root. For a leaf node split, the covering radius of the split node is set to the distance from the center to the object furthest away, that is, the actual covering radius. For an internal node split, the covering radius is not computed exactly, but over-estimated, as follows. For every child node, the covering radius of that node is added to the distance between the center of that child and that of the split node. Then, the covering radius of the split node is then set to the maximum of those sums.

Brisaboa et al. have proposed a static index structure so called the Sparse Spatial Selection (SSS) tree [9], in which the first object in a data set is selected as the first cluster center and then the rest of the objects become new cluster centers if they are far enough away from all current centers (i.e., the minimum distance between the object and current cluster centers is greater than αM, where α is a user-defined parameter and M is the maximum distance between any two objects); otherwise, they are assigned to the cluster associated with the nearest center. The process is recursively applied to those clusters that have not yet fallen below a given size threshold and the diameter M of each such cluster is estimated by using twice the covering radius of the node. Because of the clustering principle used in the construction phase, the internal nodes of SSS-trees will generally have smaller regions than the internal nodes of M-trees. However, there are still sparse regions at higher levels of SSS-trees.

Three approximate techniques for *k*-NN search were introduced by Zezula et al. [2]. The first, the so-called relative error approximation technique, controls approximation through a user-defined relative distance error $\epsilon \geq 0$. For a given query q and error ϵ, an approximation of a *k*th nearest neighbor O_A^k is called a $(1+\epsilon)$ *k*th nearest neighbor, compared to the *true* *k*th nearest neighbor O_N^k, if and only if $d(q, O_A^k) \leq (1+\epsilon) \cdot d(q, O_N^k)$. Thus, the search algorithm uses the radius $r_q/(1+\epsilon)$ instead of the covering radius of the current *k*-NN candidate set r_q to check overlap between query and data regions and candidate object qualification as well. An example of this approach is given in Figure 1a. The second, the so-called good fraction approximation technique, uses a distance distribution to provide an early termination criterion which leads to an approximate *k*NN search. In the third, the so-called small chance of improvement approximation technique, the search algorithm is based on the fact that the dynamic radius of the result set initially decreases rapidly and eventually will slow down. Thus the search stops as soon as the decrease of the radius becomes sufficient.

The PAC method [3] is an extension of $(1+\epsilon)$ nearest neighbor search by a user-specified confidence parameter $\delta \in (0, 1)$. The search algorithm stops immediately if the result satisfies the $(1+\epsilon)$ nearest neighbor with a confidence of at least δ.

Probabilistic LAESA [4] is a probabilistic technique for range search that provides user-customizable limits (θ) for the probability of false dismissals. More specifically, the radius r_q is scaled down by a factor $(1+\epsilon)$ ($\epsilon > 0$) during the filtration of indexed objects. The upper bound for $(1+\epsilon)$ is $r_q\sqrt{1-(1-\theta)^{1/p}}/(\sqrt{2}\sigma)$, where p is the number of pivots and σ^2 is the variance of the distance distribution of the data set. Figure 1b shows an example of this technique.

The region proximity technique [5] estimates the probability of the intersection of the query and data regions containing objects relevant to the query. The data region is discarded if the estimated probability is less than a user-specified threshold.

III. Our approach

First, we explain the basic principles of the so-called *best first* strategy [10] for *k*-NN search. In essence, we maintain a set of at most k (initially zero) candidates throughout the search. We also maintain a covering radius for this candidate set. This covering radius is infinite as long as we have fewer than k candidates. The algorithm processes the most promising metric regions first by maintaining a priority queue of pair of distances to regions and pointers to those regions. The following actions are repeated until the lower bound of the distance from the query to the region about to be processed is greater than the current dynamic query radius. The most promising region is popped from the queue. The objects of the current region are checked with the current dynamic query radius and, if necessary, the list of candidate *k*-NN is updated. If we have k candidate, this leads the reduction of the dynamic query radius. Those sub-regions of the current region that intersect with the query region reinserted into the queue along with the lower bound distances to them from the query.

We now look at our approach. Metric indexes may have highly sparse data regions at higher levels, with the ball radii covering large amounts of empty space—especially for high dimensionalities. During a search, in order to discard those regions that might not contain relevant objects for the query we could use any version of of the $(1+\epsilon)$ NN technique (for instance, the relative error approximation). Our suggested approach is similar to the relative error approximation technique and the key difference is to divide the *data ball radius* by $(1+\epsilon)$, rather than the query radius. See Figure 2 for an example. In the figure, we have shown what happens if we divide both the query radius and the data radius by $(1+\epsilon)$. In the first example (Figure 2a), the query lies close to the data ball, on the outside. In this case, our method lets us eliminate the region simply because it increases the lower bound more. In the second example (Figure 2b), the query is just *inside* the data ball. In this case,

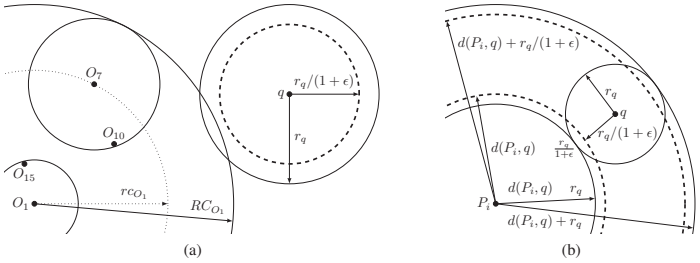


Figure 1: Examples of data and query ball regions in $\mathbb{R}^2$ with $\mathcal{L}_2$. (a) Relative error approximation in the M-tree [8] with two levels (i.e., top level with center O_1 and bottom levels with centers O_1 and O_7). RC_{O_1} represents the covering radius of top region centered at O_1 while rc_{O_1} represents its "true" covering radius. (b) Probabilistic LAESA.

shrinking the query radius will *never* lead to an elimination, whereas our method does.

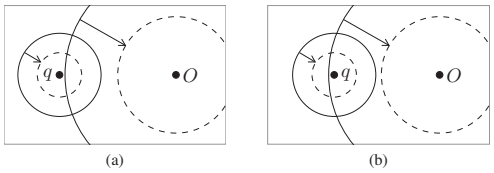


Figure 2: Examples of data and query regions in $\mathbb{R}^2$ with $\mathcal{L}_2$ (a) query q is outside the data region with center O and (b) q is inside the data region.

These examples demonstrate the twofold intuition behind our method: First, ball trees are generally built using some form of clustering. If the data set itself is clustered, and the clustering algorithm is good, this will presumably lead to the center region of a ball being more densely populated than its periphery. Even if this is not the case, by setting the radius to the maximum of all center–object distances, the radius is sensitive to outliers, and the more extreme they are, the fewer there are likely to be. Even if we do not assume a Gaussian distribution, it is not unreasonable to guess that our distance histogram will have the majority of its values clustered roughly around the mean, with fewer occurrences of very high and low distances (ignoring self-distances, $d(x,x)$). Assume that we have a global distance histogram somewhat like that in Figure 3, for example. We also assume that the center–object distances are distributed roughly according to the global distance histogram. Chávez et al. call this behavior as a "reasonable approximation" [11, p. 304]. In this case, it seems that it would be safe to shrink large data balls more than smaller ones, as the number of objects lost would be smaller. The same could be said about the smallest balls, of course; however, we would probably want to examine most of those, if they are close to the query, as they more precisely represent the objects inside them.

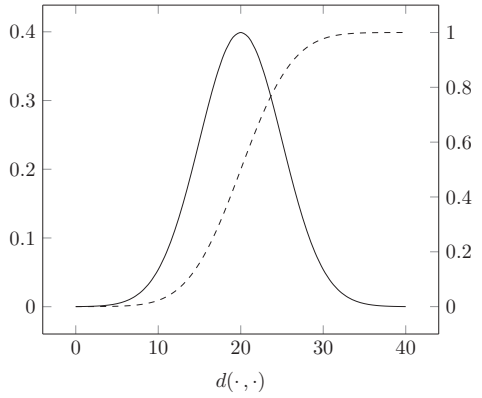


Figure 3: Distance histogram (solid) and cumulative distance histogram (dashed).

Second, shrinking the data balls affords us some elimination possibilities that simply do not exist with the original query-shrinking approach, that is, when the query falls inside the data ball. We have performed some tentative experiments to explore the relative importance of these two factors. At parameter settings that yield similar levels of error, our method generally uses fewer distance computations than the relative error method (see Section IV). We estimate that the proportion of the saved distance computations caused by cases where the query is inside the data ball to vary from about $1\,\%$ to over $50\,\%$ (data not shown).

Another contribution of this paper is that we point out and remedy a problem in the relative error approximation technique. Several experimental studies have showed that the performance of the relative error method (as originally described) is very poor [1, 2, 5]. According to Zezula et al., "the chief reason for the markedly poor performance of

the Relative Error Approximation method (with respect to the others) is that precise nearest neighbors algorithms find good candidates for the result sets soon on, and then spend the remainder of their time mostly in refining the current results" [1, p. 157]. We claim, instead, that the main reason for this performance issue is found in the pruning criterion for a candidate given object O, given by Equation 16 on page 280 of the original paper by Zezula et al. [2]:

$$\frac{r_q}{d(q,O)} < 1 + \epsilon \,,$$

or, equivalently,

$$\frac{r_q}{1+\epsilon} < d(q,O) \,,$$

where r_q is the covering radius of the current k-NN candidate set.

Now, the radius shrinking is intended to reduce the number of distance computations needed by excluding regions of low relevance. There is no need to use it here, as the distance $d(q,O)$ has already been computed, and we simply wish to know whether the object O is an improvement over the candidates we have found so far. We can determine this by simply comparing $d(q,O)$ directly to r_q. Indeed, if

$$\frac{r_q}{1+\epsilon} < d(q,O) < r_q$$

we will lose an improvement to our candidate set, involving an object whose distance we have *already computed*. This can be particularly important early on, where we wish to add good candidates (thereby reducing the dynamic search radius) as quickly as possible. In our experiments, we use this improved version of the relative error approximation technique, checking each candidate object against the actual covering radius of the candidate set.

IV. Experiments

In this section, we evaluate the performance and result quality of our technique against the amended version of the relative error approximation and the region proximity techniques on synthetic and real-world data sets. For all data sets we use the Euclidean distance.

- Uniform 10: Synthetic. 100 000 uniformly generated 10-dimensional vectors.
- Clusters 10: Synthetic. 100 000 clustered 10-dimensional vectors with 10 cluster centers. The centers were randomly chosen from a uniform distribution and objects in the clusters were generated from the multivariate normal distribution around each of the cluster centers with a variance of 0.1.
- Corel: 60 000 feature vectors with 64 dimensions extracted from the Corel image data set.
- NUS [12]: 269 648 color histograms extracted from Flickr images. Each histogram is 64-dimensional.

Amato et al. claimed that there was no practical difference between the proximity and the PAC-NN technique [5, p. 225]. Also PAC-NN is designed only for approximate NN retrieval ($k = 1$). Therefore, PAC-NN is not considered for our experiments.

We have applied our method, region proximity, and the amended version of the relative error technique on M- and SSS-trees. The maximum arities of the trees were set to 30 for the synthetic and 15 for the real-world data sets. We selected 1000 queries from the respective data set at random and the remaining objects in the data set used for indexing. We compared the search performance and result quality of three techniques by varying the result size threshold (k), using the values 1, 5, 10, 20, 40 and 80. We report only the results with 10 NN because the results with the other result size thresholds were quite similar. For the relative error approximation technique, the relative error α was varied in the interval $[0.001,\ 2.0]$ with step size 0.1 following the experimental settings of Zezula et al. [2]. For the region proximity technique, the proximity value was varied in the interval $[0.003,\ 0.06]$ with step size 0.003 following the experimental settings of Amato et al. [5]. For our technique, the data region stretching factor was varied between 0.1 and 2.0 with step size 0.1.

For each query, we counted the number of distance computations needed for the approximate search (the most commonly used criterion for measuring the performance of metric indexing structures), normalized by the number needed for an exact search, and measured a slightly modified version of the error on the position [1, 5]. The original version of the error on the position has some drawbacks. First, it gives an error value that is normalized by the size of data set. The normalized values are harder to interpret. For example, we can not directly see the absolute position difference between exact and approximate results. Second, the error should not be normalized by the approximate result's size and should take missing objects in the result set into account.

Let us have a look at a simple example: Let n be data set size and the result size threshold k be 80 ($k < n$) and the approximate result be only true 3 NNs. For some reason, the approximate result did not retrieve other 77 NNs (for instance, the search algorithm was terminated early). Then if we apply the original formula on this example, the error on the position is $((1-1)+(2-2)+(3-3))/(3n) = 0$. The error value 0 means that the approximate result has no error and as we see that it should not be 0 in this case. The modified version takes these situations into account, and the error is increased by n as a *fine* for every missing object and then the error is only normalized by k. This modified version of the error on the position yields the average absolute position difference between every point of the exact and approximate results.

In general, approximation techniques will produce results that vary both in performance and accuracy. In order to make a fair comparison between different techniques we

have to compare their speed-up factors with the same error or vice versa. In some cases, it would be difficult to achieve this goal, as neither performance measure is a deterministic function of the parameter settings. In order to compare the results properly, we plot them as a lines with one point for each parameter setting, with the coordinates for each point given by the mean error and mean normalized distance count for all queries. On the y-axis, the value 10^{-1} means that the indexing structure performed 10 times as fast as an exact search. On the x-axis, the value 10^1 means that the average absolute position difference between exact and approximate result is 10 (for instance, if the result size threshold is 1, then the 11th NN is reported instead of the NN).

In Figure 4, the errors on the position vs normalized distance counts for $10\,\mathrm{NN}$ on M-trees are shown. Note that both axis are logarithmic. For the results of NUS 10 NN in Figure 4, our technique achieved a speed-up by a factor of more than 4 over the exact search, with the position error less than 10, while the region proximity technique achieved almost same speed-up with relatively high position error 10^4 (i.e., reported 10 objects from around $10\,000$ NNs for the query). For the other data sets, our technique achieved about 2.5 speed-up over exact search with the position error less than 10.

Figure 5 shows the results for SSS-trees. The most interesting results are once again obtained with the NUS data set. For NUS 10 NN, the maximum value of the error was 48.39 (with normalized distance count 0.135) for our technique while for the region proximity technique the error was 1751.39 (with normalized distance count 0.136). The results show that our technique is faster than two other competing techniques with a low degree of the error on the position. The relative error approximation technique outperforms the region proximity technique on the real-world data sets while on the synthetic data sets it does not.

In addition to the experiments presented, we have also performed some tentative experiments involving various tradeoffs between query- and data-ball shrinking. So far, this has not yielded substantial improvements.

V. Conclusions

We have proposed an approximate similarity search technique for metric spaces and we have amended a problem in the relative error approximation technique. We have empirically evaluated our technique, showing that it outperforms the amended version of relative error approximation and the region proximity techniques.

Acknowledgements

We wish to thank Øystein Torbjørnsen and Svein Erik Bratsberg for helpful discussions.

References

[1] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search : The Metric Space Approach*. Springer, 2006.

[2] P. Zezula, P. Savino, G. Amato, and F. Rabitti, "Approximate similarity retrieval with M-Trees," *The VLDB Journal*, vol. 7, no. 4, pp. 275–293, 1998.

[3] P. Ciaccia and M. Patella, "PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces," in *Proceedings of the 16th International Conference on Data Engineering, ICDE* (IEEE Computer Society, ed.), pp. 244–255, 2000.

[4] E. Chávez and G. Navarro, "A probabilistic spell for the curse of dimensionality," in *Revised Papers from ALENEX'01*, pp. 147–160, 2001.

[5] G. Amato, F. Rabitti, P. Savino, and P. Zezula, "Region proximity in metric spaces and its use for approximate similarity search," *ACM Transactions on Information Systems, TOIS*, vol. 21, no. 2, pp. 192–227, 2003.

[6] E. Chavez Gonzalez, K. Figueroa, and G. Navarro, "Effective proximity retrieval by ordering permutations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[7] O. Edsberg and M. L. Hetland, "Indexing inexact proximity search with distance regression in pivot space," in *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, 2010.

[8] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB*, pp. 426–435, Morgan Kaufmann, 1997.

[9] N. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe, "Clustering-based similarity search in metric spaces with sparse spatial centers," in *Proceedings of SOFSEM'08*, no. 4910 in LNCS, pp. 186–197, 2008.

[10] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[11] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Computing Surveys*, vol. 33, pp. 273–321, September 2001.

[12] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y.-T. Zheng, "Nus-wide: A real-world web image database from national university of singapore," in *Proc. of ACM Conference on Image and Video Retrieval (CIVR'09)*, 2009.

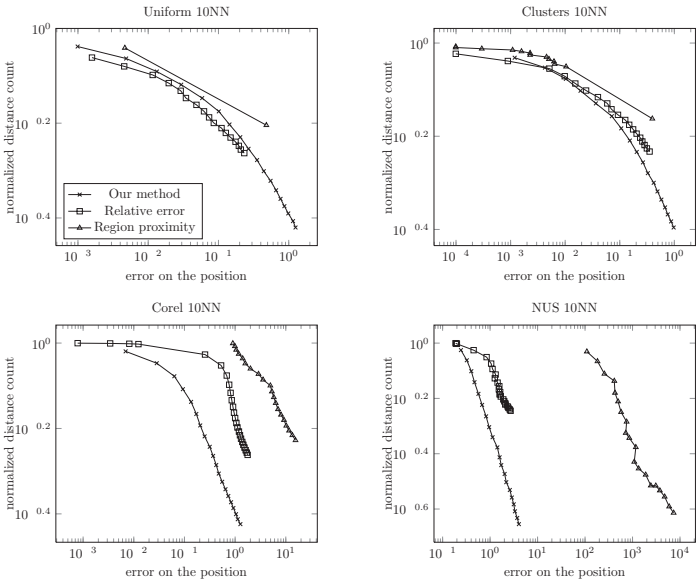


Figure 4: Performance vs. result quality of approximation on the synthetic and real-world data sets with M-trees.

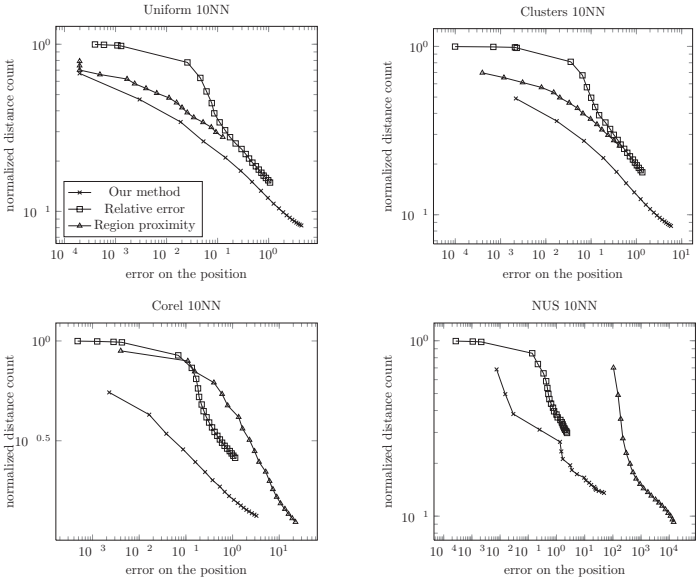


Figure 5: Performance vs. result quality of approximation on the synthetic and real-world data sets with SSS-trees.

Paper VI: Engineering Efficient and Effective Non-metric Space Library

Leonid Boytsov and Bilegsaikhan Naidan.
Appeared at The 6th International Conference on Similarity Search and Applications (SISAP), 2013.

Appendix F

Engineering Efficient and Effective Non-Metric Space Library

Leonid Boytsov[1] and Bilegsaikhan Naidan[2]

[1] Language Technologies Institute,
Carnegie Mellon University,
Pittsburgh, PA, USA
leo@boytsov.info
[2] Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway
bileg@idi.ntnu.no

Abstract. We present a new similarity search library and discuss a variety of design and performance issues related to its development. We adopt a position that engineering is equally important to design of the algorithms and pursue a goal of producing realistic benchmarks. To this end, we pay attention to various performance aspects and utilize modern hardware, which provides a high degree of parallelization. Since we focus on realistic measurements, performance of the methods should not be measured using merely the number of distance computations performed, because other costs, such as computation of a cheaper distance function, which approximates the original one, are oftentimes substantial. The paper includes preliminary experimental results, which support this point of view. Rather than looking for the best method, we want to ensure that the library implements competitive baselines, which can be useful for future work.

Keywords: benchmarks, (non)-metric spaces, Bregman divergences

1 Introduction

A lot of domains, including content-based retrieval of multimedia, computational biology, and statistical machine learning, rely on similarity search methods. Given a finite database of objects $\{o_i\}$, a search query q and a dissimilarity measure (which is typically represented by a distance function $d(o_i, q)$), the goal is to find a subset of database objects sufficiently similar to the query q.

Two major retrieval tasks are typically considered: a nearest neighbor and a range search. The nearest neighbor search aims to find the least dissimilar object, i.e., the object at the smallest distance from the query. Its direct generalization is the k-nearest neighbor (or the k-NN) search, which looks for the k most closest objects. Given a radius r, the range query retrieves all objects within a query ball (centered at the query object q) with the radius r, or, formally, all the objects $\{o_i\}$ such that $d(o_i, q) \leq r$.

The queries can be answered either exactly, i.e., by returning a complete result, or, approximately, e.g., by finding only some nearest neighbors. The exact versions of near neighbor and range search received a lot of attention. Yet, in many applications exact searching is not essential, because the notion of similarity, e.g., between two images, is not specified rigorously. Applying an exact retrieval method does not necessarily mean that we will find the image that is most similar to a query from a human perspective. Likewise, a k-NN classifier may perform well even if the search method does not produce a precise and/or a complete result [4,31].

Search methods for non-metric spaces are especially interesting. This domain does not provide sufficiently generic *exact* search methods. We may know very little about analytical properties of the distance or the analytical representation may not be available at all (e.g., if the distance is computed by a black-box device [35]). Hence, employing an approximate approach is virtually unavoidable.

Approximate search methods are typically more efficient than exact ones. Yet, it is harder to evaluate them, because we need to measure retrieval speed at different levels of recall (or any other effectiveness metric). To the best of our knowledge, there is no publicly available software suit that (1) includes state-of-the-art approximate search methods for both non-metric and metric spaces and (2) provides capabilities to measure search quality. Thus, we developed our own test framework and presented it in this paper.

1.1 Related Work

There is large body of literature devoted to exact search methods in metric spaces, which are thoroughly surveyed in the books by Faloutsos [12], Samet [32], and Zezula et al. [40] (see also a survey by Chávez et al. [5]). Exact methods have a limited value in high-dimensional spaces, which exhibit phenomena of the empty space [33] and measure concentration [5]. Experiments show that, as the dimensionality increases, every nearest neighbor search method degrades to sequential searching [39]. This is commonly known as the "curse of dimensionality". In that, methods, which are allowed to return inexact answers, are less affected by the curse [30]. For a discussion of these phenomena, we address the reader to the papers of Indyk [20] and Pestov [30].

To answer the approximate nearest neighbor queries, Indyk and Motwani [21] as well as Kushilevitz et al. [25] proposed to use random projections. The locality sensitivity hashing (LSH) is one of the most well-known implementations of this idea [21,20]. The LSH indexing uses several hash functions, such that a probability of a collision (hashing to the same value) is sufficiently high for close objects, but is small for distant ones.

The LSH works best in L_p spaces where $p \in (0, 2]$. There exists an extension of the LSH for an arbitrary metric space [28] as well as for *symmetric* non-metric distances [27]. Performance of the LSH depends on the choice of parameters, which can be tuned to fit the distribution of a data set [7].

Most exact search methods can be transformed into approximate ones by applying an *early termination strategy*. In particular, Zezula et al. [41] demon-

strated that this approach works well for M-trees. One of the most efficient strategies relies on density estimates for a distribution of distances [41,1]. The density-based approach to space pruning was also discussed by Chávez and Navarro [6] (in the context of pivoting methods), who called it "stretching" of the triangle inequality.

Let us consider a metric space, where we selected a single reference point π, known as *pivot*. The pivot is used to prune space during searching. Imagine that we computed a distance from the pivot π to every other data point. Then, points are sorted in the order of increasing distances from π. The median distance is m and points are divided into two buckets. If the distance from a point to π is smaller than m, the point is put into the first bucket. Points with distances larger than (or equal to) m are placed into the second bucket.

Let q be a query point and r be a radius of the range query. If $r < m - d(\pi, q)$, an answer can be only in the first bucket. If $r \leq d(\pi, q) - m$, an answer can be only in the second bucket. Otherwise, the answer can be in both buckets and no pruning is possible (without risking to miss an answer). In the "stretched" triangle inequality, we choose constants $\alpha_1, \alpha_2 \geq 1$.[3] If $r < \alpha_1(m - d(\pi, q))$, we check only the left bucket. If $r < \alpha_2(d(\pi, q) - m)$, we check only the right bucket. This is an example of an *oracle* procedure that defines a pruning algorithm of a pivoting method. Note that (1) it is possible to learn the oracle in both metric and *non-metric* spaces, (2) we can learn a pivot-specific oracle, instead of the global one, (3) most existing methods designed for metric spaces can be converted into non-metric search methods by simply replacing the triangle-inequality based pruning method with a search oracle. We plan to present these learning approaches in detail elsewhere.

In a recent survey [36], Skopal and Bustos discussed several types of non-metric access methods, which we divide into the following categories: (1) projective and lower/upper bounding approaches, (2) methods that prune the space using properties other than the triangle inequality (e.g., the Ptolemaic inequality [26]), and (3) domain-specific methods. Inverted files are a classic domain-specific algorithm applicable to high-dimensional, but sparse, vector spaces, where the distance function is the cosine similarity (or a similar distance).

Jacobs et al. [22] review various projection methods and argue that a projection is not always feasible, for instance, when the similarity cannot be expressed by a numeric distance function, or the distance function is not symmetric. In the case of symmetric, non-negative, and reflexive distance, one can use the TriGen algorithm [35], which applies a monotonic transformation to the distance function. Consider, e.g., the squared Euclidean distance, which is a (non-metric) Bregman divergence. By taking the square root, we obtain the metric function. Similarly, the TriGen algorithm allows one to convert a distance into a function that satisfies the triangle inequality only approximately. In addition, it provides control over the degree of approximation.

Chávez et al. [18] proposed a projective method, which is applicable to both metric and *non-metric spaces*. The method, called the permutation index, selects

[3] Chávez and Navarro [6] employed only one stretching constant.

k pivots $\{\pi_i\}$ and for every data point o it creates a permutation of pivots: a list where pivots are sorted in the order of increasing distances $d(\pi_i, o)$. Independently, this method was invented by Amato and Savino [2], who additionally proposed to index permutations using an inverted file.

To answer the query, the correlation is computed between the permutation of the vector and the permutation of every data point. Then, all data points are sorted in the order of ascending correlation values and a given fraction of objects are compared directly with the query (by computing the distance in the original space). Performance of the permutation index can be improved by using incremental sorting [17] or by indexing permutations using an inverted file [2], a permutation prefix tree [11], or a metric space index [14].

Bregman divergences is a class of non-metric distance functions. This divergences include the squared Euclidean distance, the KL-divergence:

$$d(x, y) = \sum x_i \log(x_i/y_i) \tag{1}$$

and the Itakura-Saito distance:

$$d(x, y) = \sum x_i/y_i - \log(x_i/y_i) - 1. \tag{2}$$

For the Bregman divergences, there exist two exact search methods. The Bregman ball tree (bbtree) [4], which recursively divides the space using two covering Bregman balls at each recursion step, and a mapping method due to Zhang et al. [42]. Both approaches use properties of Bregman divergences to lower/upper bound distance values.

2 Methodology

2.1 Evaluation Approach

Performance of approximate methods is typically represented by a curve that plots efficiency against effectiveness. Two most common efficiency metrics are retrieval time and a number of distance computations. Additionally, we use the *improvement in efficiency* (with respect to the single-thread sequential search algorithm) and the *improvement in the number of distance computations*.

Recall is a commonly used effectiveness metric. It is equal to the fraction of all correct answers retrieved. The *relative error* [41] is defined for a pair of points o and $\widetilde{o}$, such that o is an exact and $\widetilde{o}$ is an approximate answer. It is simply a ratio of the distances $d(\widetilde{o}, q)$ and $d(o, q)$. The relative error can be misleading, especially in high dimensional spaces. Due to high concentration of measure, an increase in relative error can be very small, but the method can return the 1,000th nearest-neighbor instead of the most closest one. This concern was also expressed by Cayton [4]. Similarly, recall does not account for position information and has the same issue [1].

Let $\text{pos}(o_i)$ represent a positional distance from o_i to the query, i.e., the number of objects closer to the query than o_i plus one. In the case of ties, we

assume that the object with a smaller index is closer to the query. Note that $\text{pos}(o_i) \geq i$. A *relative position* error is equal to $\text{pos}(o_i)/i$ and is more informative than a relative distance error and/or recall. We average relative position errors using the geometric mean [23].

Zezula et al. [41] proposed to use the average value of the inverse relative position error (called the precision of approximation) as a performance metric (m is the number of found objects):

$$\frac{1}{m}\sum_{i=1}^{m}\frac{i}{pos(o_i)} \tag{3}$$

Amato et al. [1] suggested the metric that measures the absolute position error. It is equal to:

$$\frac{1}{m}\sum_{i=1}^{m}\frac{pos(o_i)-i}{\#\text{of indexed points}} \tag{4}$$

Unfortunately, this metric produces results that are not comparable across collections and result sets of different sizes. Consider an example of the result set, where $pos(o_i) = 2i$. The absolute position error is equal to:

$$\frac{1}{m}\sum_{i=1}^{m}\frac{2i-i}{\#\text{of indexed points}} = \frac{0.5(m+1)}{\#\text{of indexed points}}$$

We have no good explanation why the position error should grow with m, while the relative position error and the degree of approximation remain constant (in this case). Even worse, due to the large factor in the denominator of Eq. 4, the computed error is generally very small. It is easy to make a wrong conclusion that the algorithm works almost ideally, whereas, in truth, it provides a poor approximation.

If we have a separate test set, testing is straightforward. Otherwise, we need to randomly divide the original data set into indexable data and testing data. This method is based on the assumption that distributions of test queries and indexed data objects are similar. The random division should be repeated several times (an approach known as bootstrapping), and performance metrics computed for each split should be aggregated.

One may be tempted to select queries among indexed data objects, or, alternatively, to create test vectors by randomly perturbing the indexed data. Both approaches are not ideal and can lead to overly optimistic or pessimistic results, especially, in the case of the nearest neighbor searching. We experimented with the `Colors` data set[13], indexed using the Vantage Point tree (VP-tree) [37]. If we selected queries from the vectors that were already indexed, it took on average only 20 distance computations to find the query's nearest neighbor. Since the query and the found vector were identical, the pruning algorithm was unrealistically efficient. For the randomly selected held-out test data, it took about 6,000 distance computations to answer the nearest neighbor query! If we used a query obtained by random additive (and uniform) perturbations of vector elements, the results depended on the amount of noise. In our experiment, we got

800 distance computations in one case and 10^5 distance computations (i.e., the algorithm degraded to the linear scan) in another.

We speculate that queries obtained by random perturbations can be useful if the model of random perturbations fits data well. This assumption is apparently reasonable for the Euclidean data, but additive transformations may significantly change the histogram of distances in the case of the KL-divergence. In one example, the application of the additive noise led to a 2x decrease in the median distance value between two randomly selected vectors. The *multiplicative log-normal* noise seemed to produce more realistic results, yet, additional experimentation is needed to understand applicability of this approach.

2.2 Choice of Programming Language

C,C++, and Java are the three most popular general-purpose programming languages[24]. [4] The authors are familiar with all three and considered them as implementation languages. According to "The Computer Language Benchmarks Game", C and C++ have comparable performance.[5] Major C/C++ compilers (GNU C++ and Microsoft Visual C) support Single Instruction Multiple Data (SIMD) commands, which allows one to compute distances more efficiently.

Yet, only C++ supports run-time and compile-time polymorphism. The new C++ specifications standardize multi-threading and simplify the use of STL containers (threads are not standardized in the pure C). [6] There is evidence, including anecdotal experience of authors, that C++ allows programmers to be more productive than does C [3]

Even though performance of Java sometimes matches performance of C++ [38,34], Java is generally 2-3 times slower than C or C++ [19,16]. Unlike C/C++, there is no built-in support for the SIMD instructions [29]. Java objects are heavy and programmers have to use parallel arrays as well as manual memory management (e.g., reusing small objects) to work around this problem [10]. Thus, writing "algorithmic-intensive" applications in Java may sometimes be harder than in C++.

Because C++ is largely a superset of C, reusing the code already implemented in the Metric Spaces Library would be straightforward. Yet, it is harder to port C-code to Java. There are tools for seamless integration of C++ and R. In particular, one can call R scripts directly from a C++ program [9]. All in all, using the latest C++ compiler that implements the new standard is the most appealing choice for us.

[4] See, also `http://www.langpop.com/` and `http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages`

[5] According to at least this page: `http://benchmarksgame.alioth.debian.org/u64/benchmark.php`

[6] See `http://www.open-std.org/jtc1/sc22/wg21/`

2.3 Design

Our software was designed in the spirit of the Metric Spaces Library [13], but there are multiple important differences. We have classes that represent an `Object`, a `Space`, and an `Index`. The `Space` abstraction is necessary to encapsulate the computation of the distance. We can have multiple `Space` sub-classes implementing different distance functions. In addition, the `Space` class provides functionality to read objects from a file.

A distance can be integer-valued, real-valued, or represented by an arbitrarily complex object (if, e.g., we compare objects using multiple criteria). Similarly to the Metric Spaces Library, the same implementation can work with different distance types (e.g., the VP-tree can be used with both the integer-valued edit distance and with the real-valued L_1 metric). This is effectively supported by the compile-time polymorphism of C++ (templates). All implementations (including indices for real-valued and integer-valued distances) co-exist in the same binary and there is no need to update makefiles when a new method or a distance is implemented.

The `Object` has an identifier and can store arbitrary data (of any type). When necessary objects are transformed: One may need to reduce the dimensionality or precompute the logarithms to accelerate evaluation of the KL-divergence (see Section 2.4). Unlike the Metric Spaces Library, a distance function accepts pointers to the objects rather than object ids.

A `Query` object proxies distance evaluations during search time, which allows us to get the average number of computations carried out by a search method as well as to compute confidence intervals (even in the *multi-threading* testing mode). It is still possible to access the distance function through a `Space` object, but this should be done only at indexing time. If (due to programmer's error) an instance of the `Index` tried to access distance through the `Space` object, the program would terminate. [7]

There are two types of query classes and both classes have the `Radius` function. For the range queries, this function returns the constant value specified by the user. For the k-NN queries, the value returned by `Radius` changes during the search, because it represents the distance from the query to the k-th closest object found so far. Because of this abstraction, it is often sufficient to implement a single (template) function that handles both the range and the k-NN search.

The distance function can be non-symmetric, thus two types of queries (left and right are possible). Currently, the framework directly supports only the left queries (q is the second argument of the distance function). For some methods, e.g., permutation-based approaches, right queries can be implemented by simply swapping arguments of the distance function. Yet, a different distance function as well as a transformation of original data may be necessary for Bregman divergences [4]. We plan to address this issue in the future.

[7] We actually have two versions of the `Space` distance function. The public, restricted, version "knows" the current phase (indexing or searching). It terminates the program if called during the search phase. The unrestricted, but private, function is accessible by a `Query` object at search time, because the latter is a friend of `Space`.

As explained in Section 1.1, one can use the concept of the search oracle to convert metric access methods into non-metric ones. We implement two oracle classes (one is based on sampling and another on "stretching" of the triangle inequality). Currently, only the VP-tree can use the generic search oracle, but we plan to embed the search oracle into other metric indices. In addition, most tree-based methods in our library implement a simple early termination strategy, where the search stops after visiting a given number of buckets.

There is a special function that governs the test process. It creates a `Space` object, which loads a data set into memory, divides the data into testing and training sets or loads a separate test set (see Section 2.1). Then, the `Factory` creates instances of specific methods. Parsing of method-specific command line parameters, though, is delegated to the `Index`. Search methods explicitly return pointers/ids of found objects. Thus, we can verify methods' correctness, as well compute recall and other effectiveness metrics discussed in Section 2.1.

Because there are no exact search methods for generic non-metric spaces, evaluation involves comparing a query object against every object in the database. This expensive procedure can be optimized: When we test several different methods in a single session, we compute exact answers only once for each query object. This is reasonably fast on our current data sets, but in the future we may memorize answers, so that they can be re-used when we run multiple tests (using the same data set).

The testing module saves evaluation results to a CSV-file and produces a human readable report. Note that the plots in Section 3 are produced by a Python script that read and processed such CSV-files.

We decided to focus on memory-resident indices. On one hand, modern servers have plenty of memory and a typical high-performance search application would keep most of its index in memory. On the other hand, we do not have to implement serialization/de-serialization or, the code that searches data stored on disk. This simplification allows us to be more productive coders. Our implementations create essentially static indices from scratch. In the future, we plan to consider incremental indexing approaches as well.

One purpose of serialization is to estimate space requirements. Yet, it is possible to obtain an approximate size of the index by measuring the amount of memory used by the program before and after the index is created (one should also include memory to store data objects). There is an opinion that better estimates can be achieved, if we compute the size of allocated memory ourselves, by writing the special code that traverses the index and measures the size of atomic index elements (such as vectors). Yet, we believe that this approach is error prone.

2.4 Efficiency Issues

Even though some distance functions are expensive, it can be quite cheap to compare two vectors using an L_p norm. Furthermore, it can be done even faster using special SIMD instructions [15]. Currently, most x86 CPUs support operations with 128-bit registers containing, e.g., 4 single-precision or 2 double-precision

Table 1: The number of computations per second for optimized and unoptimized distance functions.

	128 elements				1024 elements			
Distance	L_1	L_2	Itakura-Saito	KL-div.	L_1	L_2	Itakura-Saito	KL-div.
C++ (no logs)	$9.6 \cdot 10^6$	$9.1 \cdot 10^6$	$1.9 \cdot 10^5$	$5.3 \cdot 10^5$	$1.2 \cdot 10^6$	$1.2 \cdot 10^6$	$2.4 \cdot 10^4$	$6.7 \cdot 10^4$
SIMD (precomp. logs)	$2.7 \cdot 10^7$	$3.3 \cdot 10^7$	$8.3 \cdot 10^6$	$2.8 \cdot 10^7$	$3.4 \cdot 10^6$	$4.5 \cdot 10^6$	$1.04 \cdot 10^6$	$2.4 \cdot 10^6$

Note: vector elements are randomly, uniformly, and independently drawn from $(0, 1]$

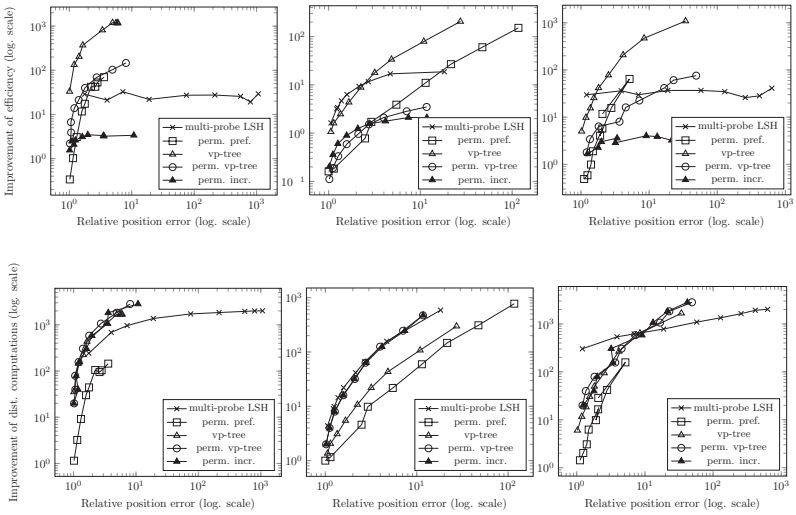
numbers. Some CPUs already support operations with 256-bit registers, which can process 8-element vectors of single-precision numbers.[8] This fact is rather well known, but it appears to be underappreciated. In addition, evaluation of some distance functions can be accelerated at the expense of higher storage requirements (or by dimensionality reduction). In the case of the KL-divergence and the Itakura-Saito distance we can precompute and memorize logarithms of vector elements.

According to Table 1, a single CPU core can carry out more than 30 million computations of the Euclidean distance between two 128-element vectors and more than 4 million distance computations between two 1024-element vectors. In that, the efficient SIMD version spends about one CPU cycle per vector element. [9] The optimized versions of the L_1, L_2 and distances, which use SIMD, are 3 times faster than pure C++ versions. The optimized versions of the KL-divergence and of the Itakura-Saito distance are about 30-50 times as fast as the original ones. In comparison, for a data set of dimensionality 128, the bbtree, which is designed to search using the KL-divergence, is only 5 times faster than sequential scan [4]. It should now be clear that (1) distance computations are not necessarily expensive and (2) optimizing computation of the distance function can be more important than designing data structures.

It has been claimed that a random memory access may take hundreds of CPU cycles [8]. Yet, our experiments showed the cost of a random access on our server to be only 60 cycles. Thus, reducing memory fragmentation may not necessarily lead to substantial improvements in performance. In particular, storing vectors of a VP-tree bucket in adjacent memory regions did not allow us to get more than a 2x speedup. Perhaps, more importantly, the SIMD-based algorithms of distance computations are so fast that communications with RAM can become a major bottleneck in a multi-threading environment. Indeed, to sustain the processing speed of one vector element per CPU cycle (see Table 1) we need to read from memory at the speed of $\approx$ 12 GB/sec (one element is a 4-byte single-precision number). Our server's memory bandwidth of 20 GB/sec can be exhausted with just two threads.

[8] See, e.g., `http://software.intel.com/en-us/avx`

[9] Reading unaligned data does not apparently hurt performance, even for SIMD operations.



(d) Colors, L_2 (e) Unif-64, L_2 (f) RCV-128, L_2

Fig. 1: Improvement in efficiency and in the number of distance computations for 1-NN search in L_2.

3 Experiments

Experiments were carried out on a Linux server equipped with Intel Core i7 2600 (3.40 GHz, 8192 KB of L3 CPU cache) and 16 GB of DDR3 RAM (transfer rate 20GB/sec). The code was compiled using GNU C++ 4.7 (optimization flag -Ofast) and tested in a single-thread (using 1,000 queries). The library can be downloaded from GitHub.[10]

The following collections were used:

1. `Colors`: 112-dimensional data set from the Metric Spaces Library [13];
2. `Unif64`: 64-dimensional vectors with elements generated randomly, independently, and uniformly;
3. `RCV-16` and `RCV-128`: 16- and 128-dimensional topic histograms [4];
4. `SIFT`: the normalized 1111-dimensional SIFT signatures [4].

We extracted the first 10^5 vectors from collections (1)-(3) and used the whole collection (4), which contained only 10^4 vectors.

We carried out two series of experiments (both involving 1-NN search). In the first series (see Fig. 1), we used collections `Colors`, `Unif-64`, and `RCV-128`. The distance was Euclidean. We measured both the improvement in efficiency and in the number of distance computations. The values of efficiency metrics were

[10] `https://github.com/searchivarius/NonMetricSpaceLib`

plotted against the relative position error. In the second experimental series (see Fig. 2), we measured how the improvement in efficiency corresponded to the relative position error. Two Bregman divergences were used: the KL-divergence (see Eq. 1) and the Itakura-Saito distance (see Eq. 2). Implemented methods included domain specific and permutation-based approaches as well the VP-tree.

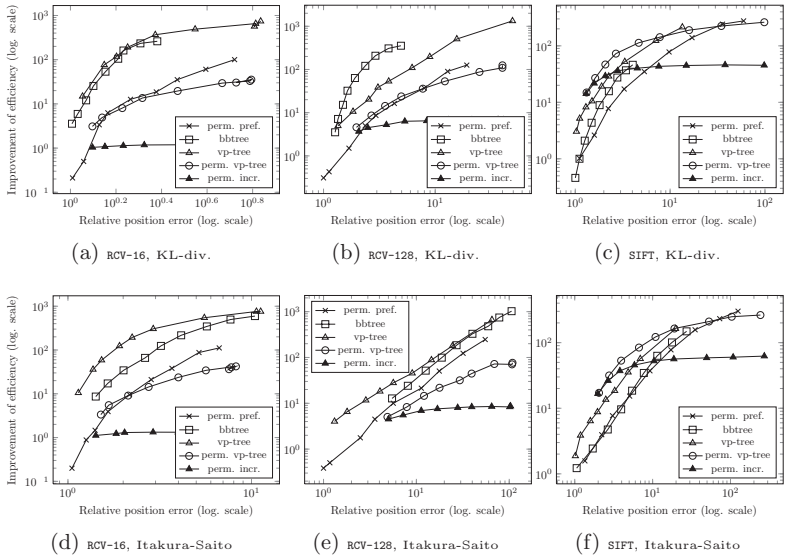
- The VP-tree employed the search oracle that "stretched" the triangle inequality (see Section 1.1). Optimal stretching coefficients were found using a simple grid search. We indexed a small database sample ($\approx$ 1,000 vectors), executed the 1-NN search for various values of stretching coefficients and measured performance. Then, we selected coefficients resulting in the fastest search at given recall values.
- Permutation-based approaches were: an improved permutation index with incremental sorting [17], a permutation prefix tree [11], and the method where permutations were indexed using a metric space index, as proposed by Figueroa and Fredriksson [14]. Unlike Figueroa and Fredriksson, we used an approximate method (the VP-tree that stretched the triangle inequality using $\alpha_1 = \alpha_2 = 2$). In all cases, we used 16 pivots and the prefix length was 4. The maximum fraction of the objects exhaustively compared against the query depended on the data set and varied from 0.01 to 0.05. The minimum fraction of the database objects to be scanned was 0.0002. The number of candidate objects in the permutation prefix index varied from 1 to 24,000.
- The bbtree [4] is the exact indexing method for Bregman divergences. It was extended by the early termination strategy, where the search stopped after visiting a certain number of buckets (the number varied from one to 1,000).
- The multi-probe LSH is designed only for L_2. We used the LSHKit implementation with the following parameters: $H = 1017881$, $T = 10$, $L = 50$.[11]

All methods, including the multi-probe LSH, relied on optimized distance functions. The correlation function (Spearman's rho) was also optimized and implemented using SIMD instructions. The vectors in the buckets of the VP-tree and bbtree were stored in contiguous chunks of memory (the bucket size was 50).

From Fig. 1 we learn that both the classic permutation method (without the index over permutations) and the multi-probe LSH carried out fewer distance computations than most other methods. Yet, they were generally outperformed by the VP-tree and the methods that index permutations (using either the prefix tree or the VP-tree). The reason is that exhaustive comparison of data-object permutations against the permutation of the query vector is costly. In that, the permutation index worked better for high-dimensional data (see Fig. 2f and 2c). Again, we see that the number of distance computations is not necessarily a good predictor of method's performance. Yet, it may give insights into scalability of methods with respect to the size of the data set and data dimensionality.

As can be seen from Fig. 2, we implemented strong baselines that worked well in *non-metric* spaces with *non-symmetric* distance functions. Note that the bbtree, which was tailored to spaces with Bregman divergences, was outperformed

[11] Remaining parameters were automatically computed by the LSHKit [7].



(a) RCV-16, KL-div. (b) RCV-128, KL-div. (c) SIFT, KL-div.

(d) RCV-16, Itakura-Saito (e) RCV-128, Itakura-Saito (f) SIFT, Itakura-Saito

Fig. 2: Improvement in efficiency of 1-NN search for the KL-divergences and Itakura-Saito distance.

by the VP-tree (which is a generic method) in most cases. These are encouraging results, but more work needs to be done. We plan to employ new complex domains and implement additional search methods.

Acknowledgments. We would like to thank Lawrence Cayton for providing the data sets, Vladimir Pestov for the discussion on the curse of dimensionality, and anonymous reviewers for helpful suggestions.

References

1. Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region proximity in metric spaces and its use for approximate similarity search. ACM Trans. Inf. Syst. **21**(2) (April 2003) 192–227
2. Amato, G., Savino, P.: Approximate similarity search in metric spaces using inverted files. In: Proceedings of the 3rd international conference on Scalable information systems. InfoScale '08, ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2008) 28:1–28:10
3. Bhattacharya, P., Neamtiu, I.: Assessing programming language impact on development and maintenance: a study on C and C++. In: Software Engineering (ICSE), 2011 33rd International Conference on. (2011) 171–180

4. Cayton, L.: Fast nearest neighbor retrieval for bregman divergences. In: Proceedings of the 25th international conference on Machine learning. ICML '08, New York, NY, USA, ACM (2008) 112–119
5. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.L.: Searching in metric spaces. ACM Computing Surveys **33**(3) (2001) 273–321
6. Chávez, E., Navarro, G.: Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. Information Processing Letters **85**(1) (2003) 39–46
7. Dong, W., Wang, Z., Josephson, W., Charikar, M., Li, K.: Modeling lsh for performance tuning. In: Proceedings of the 17th ACM conference on Information and knowledge management. CIKM '08, New York, NY, USA, ACM (2008) 669–678
8. Drepper, U.: What every programmer should know about memory (2007) `http://www.akkadia.org/drepper/cpumemory.pdf` [Last checked August 2012].
9. Eddelbuettel, D., Francois, R.: Rcpp: Seamless R and C++ integration. Journal of Statistical Software **40**(8) (4 2011) 1–18
10. Elizarov, R.: Millions quotes per second in pure Java (2013) `http://blog.devexperts.com/millions-quotes-per-second-in-pure-java/` [Last accessed on May 14th 2013].
11. Esuli, A.: Use of permutation prefixes for efficient and scalable approximate similarity search. Inf. Process. Manage. **48**(5) (September 2012) 889–902
12. Faloutsos, C.: Searching Multimedia Databases by Content. Kluwer Academic Publisher (1996)
13. Figueroa, K., Navarro, G., Chávez, E.: Metric Spaces Library (2007) Available at `http://www.sisap.org/Metric_Space_Library.html`.
14. Figueroa, K., Fredriksson, K.: Speeding up permutation based indexing with indexing. In: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications. SISAP '09, Washington, DC, USA, IEEE Computer Society (2009) 107–114
15. Fredriksson, K.: Engineering efficient metric indexes. Pattern Recognition Letters **28**(1) (2007) 75 – 84
16. Fulgham, B.: The computer language benchmarks game (2013) `http://benchmarksgame.alioth.debian.org/` [Last accessed on May 14th 2013].
17. Gonzalez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(9) (2008) 1647–1658
18. Gonzalez, E.C., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(9) (2008) 1647–1658
19. Hundt, R.: Loop recognition in C++/Java/Go/Scala. Proceedings of Scala Days **2011** (2011)
20. Indyk, P.: Nearest neighbors in high-dimensional spaces. In Goodman, J.E., O'Rourke, J., eds.: Handbook of discrete and computational geometry. Chapman and Hall/CRC (2004) 877–892
21. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing. STOC '98, New York, NY, USA, ACM (1998) 604–613
22. Jacobs, D., Weinshall, D., Gdalyahu, Y.: Classification with nonmetric distances: Image retrieval and class representation. Pattern Analysis and Machine Intelligence, IEEE Transactions on **22**(6) (2000) 583–600
23. King, G.: How not to lie with statistics: Avoiding common mistakes in quantitative political science. American Journal of Political Science (1986) 666–687

24. King, R.S.: The top 10 programming languages [the data]. Spectrum, IEEE **48**(10) (2011) 84–84
25. Kushilevitz, E., Ostrovsky, R., Rabani, Y.: Efficient search for approximate nearest neighbor in high dimensional spaces. In: Proceedings of the 30th annual ACM symposium on Theory of computing. STOC '98, New York, NY, USA, ACM (1998) 614–623
26. Lokoč, J., Hetland, M.L., Skopal, T., Beecks, C.: Ptolemaic indexing of the signature quadratic form distance. In: Proceedings of the Fourth International Conference on SImilarity Search and APplications. SISAP '11, New York, NY, USA, ACM (2011) 9–16
27. Mu, Y., Yan, S.: Non-metric locality-sensitive hashing. In: AAAI. (2010)
28. Novak, D., Kyselak, M., Zezula, P.: On locality-sensitive indexing in generic metric spaces. In: Proceedings of the Third International Conference on SImilarity Search and APplications. SISAP '10, New York, NY, USA, ACM (2010) 59–66
29. Parri, J., Shapiro, D., Bolic, M., Groza, V.: Returning control to the programmer: Simd intrinsics for virtual machines. Commun. ACM **54**(4) (April 2011) 38–43
30. Pestov, V.: Indexability, concentration, and {VC} theory. Journal of Discrete Algorithms **13**(0) (2012) 2 – 18 Best Papers from the 3rd International Conference on Similarity Search and Applications (SISAP 2010).
31. Pestov, V.: Is the k-NN classifier in high dimensions affected by the curse of dimensionality? Computers & Mathematics with Applications (2012)
32. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc. (2005)
33. Scott, D.W., Thompson, J.R.: Probability density estimation in higher dimensions (1983) Technical Report, Rice University, Texas Huston.
34. Shafi, A., Carpenter, B., Baker, M., Hussain, A.: A comparative study of java and c performance in two large-scale parallel applications. Concurrency and Computation: Practice and Experience **21**(15) (2009) 1882–1906
35. Skopal, T.: Unified framework for fast exact and approximate search in dissimilarity spaces. ACM Trans. Database Syst. **32**(4) (November 2007)
36. Skopal, T., Bustos, B.: On nonmetric similarity search problems in complex domains. ACM Comput. Surv. **43**(4) (October 2011) 34:1–34:50
37. Uhlmann, J.: Satisfying general proximity similarity queries with metric trees. Information Processing Letters **40** (1991) 175–179
38. Vivanco, R.A., Pizzi, N.J.: Scientific computing with Java and C++: a case study using functional magnetic resonance neuroimages. Software: Practice and Experience **35**(3) (2005) 237–254
39. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of the 24th International Conference on Very Large Data Bases, Morgan Kaufmann (August 1998) 194–205
40. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach (Advances in Database Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
41. Zezula, P., Savino, P., Amato, G., Rabitti, F.: Approximate similarity retrieval with m-trees. The VLDB Journal **7**(4) (December 1998) 275–293
42. Zhang, Z., Ooi, B.C., Parthasarathy, S., Tung, A.K.H.: Similarity search on bregman divergence: towards non-metric indexing. Proc. VLDB Endow. **2**(1) (August 2009) 13–24

Paper VII: An empirical evaluation of a metric index for approximate string matching

Bilegsaikhan Naidan and Magnus Lie Hetland.
Appeared at Norsk Informatikkonferanse (NIK), 2012.

Appendix G

An empirical evaluation of a metric index for approximate string matching

Bilegsaikhan Naidan and Magnus Lie Hetland

Abstract

In this paper, we evaluate a metric index for the approximate string matching problem based on suffix trees, proposed by Gonzalo Navarro and Edgar Chávez [9]. Suffix trees are used during the index construction to generate intermediate data (pivot table) that to be indexed and the query processing. One of the main problems with suffix trees is their space requirements. To address this, we proposed as an alternative a linear-time algorithm that simulates suffix trees in the suffix arrays. The proposed algorithm is more space-efficient and is more suited for disk-based implementation. Even so, experimental results on two real-world data sets show that the metric index is beaten by straightforward, slightly enhanced linear scan.

1 Introduction

Approximate string matching is crucial for many modern applications, such as in computational biology. The main goal of this problem is to find all the occurences of a given query string in a large string permitting a given amount of error in the matching.

Nowadays, the metric space model is becoming a favored approach for solving many approximate or distance-based search problems. Given a metric d over a universe $\mathbb{U}$, and a data set $\mathbb{D} \subset \mathbb{U}$, the task is to quickly retrieve the objects in $\mathbb{D}$ that are within a given search radius (or the k nearest neighbors) of some query $q \in \mathbb{U}$ according to the metric d. For strings, a metric d can be the edit distance (or Levenstein distance), which gives the minimum number of insertions, deletions, and replacements required to transform one string into another. The edit distance between strings x and y can be computed in $\mathcal{O}(|x| \cdot |y|)$ time and $\mathcal{O}(\min(|x|, |y|))$ space. Several approaches [8] have been proposed to improve the efficiency of approximate string matching. However, most of these approaches are only focused on short string matching. In DNA sequences, it is necessary to align long queries in a large sequence. For more details on various approaches to DNA sequence alignment, see the recent survey by Li and Homer [7] and the comparative analysis by Ruffalo et al. [10].

Gonzalo Navarro et al. [9] proposed a metric indexing structure for approximate string matching based on suffix trees. According to Kurtz [6], improved implemen-

This paper was presented at the NIK-2012 conference; see http://www.nik.no/.

tations of linear-time suffix tree construction algorithm require 20 times more space than the input string in the worst case. For instance, the human genome is about 3 GiB of symbols and its suffix tree requires about 60 GiB space. Thus it may not fit in the main memory of many systems. Also, disk-based implementation of suffix trees is not straightforward. A more space-efficient data structure is the *suffix array*. To summarize, the main contributions of this paper are outlined as follows:

- We propose a more efficient algorithm that simulates the index construction process, using suffix arrays.
- We have conducted experiments evaluating the metric index structure, providing empirical results. (The original authors provided only theoretical results.)

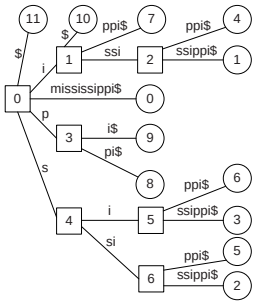
The remainder of this paper is organized as follows. In Section 2, some notation and preliminary definitions are given. Section 3 presents the original version of the metric index while Section 4 describes our algorithm. Experimental results are provided in Section 5 and finally Section 6 concludes the paper.

2 Preliminaries

In this section we introduce some notation and definitions that are used in the rest of the paper.

Let Σ be a finite ordered alphabet and let S be a string over Σ. We assume that S ends with a special end symbol \$ that is not included in Σ and lexicographically ordered before any symbol in Σ. The length of S is denoted by $n = |S|$. A substring $S[i \dots j]$ of S starts at position i and ends at position j ($0 \leq i \leq j < n$). For simplicity, we let $S[i]$ denote $S[i \dots n-1]$ ($0 \leq i < n$).

The suffix tree $\mathcal{T}$ for S is a tree that compactly represents all the suffixes of S and has exactly n leaves. The leaves contain unique integers in the range $[0, n-1]$ that indicate the starting positions of the suffixes. Each edge of the tree is labeled with a substring of S, so that the concatenated edge labels on a path from root to leaf form the suffix represented by the given leaf node. An example of a suffix tree for string "mississippi\$" is shown in Figure 1a. For more detailed explanation see book by Gusfield [3].



(a)

i	$sa[i]$	$lcp[i]$	$S[sa[i]]$
0	11	0	\$
1	10	0	i\$
2	7	1	ippi\$
3	4	1	issippi\$
4	1	4	ississippi\$
5	0	0	mississippi\$
6	9	0	pi\$
7	8	1	ppi\$
8	6	0	sippi\$
9	3	2	sissippi\$
10	5	1	ssippi\$
11	2	3	ssissippi\$

(b)

Figure 1: Illustrations of a suffix tree for string "mississippi\$" (a) a suffix array for the same string and its lcp array (b).

The suffix array *sa* is an integer array which contains the starting positions of lexicographically ordered suffixes of S. The longest common prefix array *lcp* is an

array of integers in which $lcp[0]$ is initialized to 0 and $lcp[i]$ indicates the length of the common prefix of $S[sa[i]]$ and $S[sa[i-1]]$, for $1 \leq i < n$. Figure 1b shows an example of a suffix and a lcp array for string "mississippi$". The average value of lcp will be $\mathcal{O}(\log n)$ [1].

We use the notation $x[y]$ from the original work by Navarro et al. [9], which represents the group of substrings $\{xy_1, xy_1y_2, ..., xy\}$ on an explicit node of suffix tree, where the current node's parent corresponds to the string x and the label of current node corresponds to the string y. For instance, the explicit internal node $i(2)$ in Figure 1a represents the strings "i[ssi]" = {"is", "iss", "issi"} and the external node $e(8)$ represents the strings "p[pi]" = {"pp", "ppi"}. Let $\mathcal{I}$ be the set of all the strings of explicit internal nodes of $\mathcal{T}$. For the string "mississippi$", $\mathcal{I}$ is {"[i]", "i[ssi]", "[p]", "[s]", "s[i]", "s[si]"}. Let $\mathcal{E}$ be a set of all the strings of external nodes of $\mathcal{T}$. Let $\mathcal{E}^*$ be $\mathcal{E} \setminus \mathcal{I}$. For the string "mississippi$", the substring "[i]" of $i(1)$ is already in $\mathcal{I}$, and thus "i" of $e(10)$ is not included in $\mathcal{E}^*$. Therefore, $\mathcal{E}^*$ is {"$", "i[ppi]", "issi[ppi]", "issi[ssippi]", "[mississippi]", "p[i]", "p[pi]", "si[ppi]", "si[ssippi]", "ssi[ppi]", "ssi[ssippi]"}.

3 The metric index

As can be seen in Section 1, some distance functions (e.g., the edit distance) are computationally expensive and to answer similarity queries by performing a linear scan on large data sets is impractical. Thus we usually build an index structure over the data set to reduce the overall query processing costs by exploiting the triangle inequality.

Most metric indexing methods are based on a filter-refine principle. One important example is so-called *pivot*-based methods, where a set of reference objects (the pivots) are selected from the dataset, and the distances to these form coordinates in a *pivot space*. In other words, the objects of the original space are embedded into the pivot space by computing the distances between pivots and those objects. Some of these distances are stored to reduce the number of distance computations during query processing. As the number of pivot increases, query processing in the pivot space becomes expensive. We note that query processing in the transformed space should be cheaper than the original one, otherwise, this entire effort is useless. In the filtration step, objects that can not qualify as relevant to a query are filtered out by establishing lower bounds of the real distances; this is done by using the pre-computed distances together with the query-pivot distances and the triangle inequality. If the lower bound is greater than the query radius the object can safely be filtered out. In the refinement step, the real distances between the query and the candidate objects obtained from the previous step are computed, and the objects that qualify as relevant reported.

A naïve approach of metric indexing for S is to build an object-pivot distance table over all the $\mathcal{O}(n^2)$ substrings of S, which is unacceptable for large strings. Thus, Navarro et al. [9] proposed an indexing algorithm that uses suffix trees during index construction and query processing. With suffix trees, we collapse all the $\mathcal{O}(n^2)$ substrings of S into the $\mathcal{O}(n)$ strings of explicit suffix tree nodes (i.e., $\{\mathcal{I} \cup \mathcal{E}^*\}$) as well as speeding up the computation of the edit distance between pivots and $\{\mathcal{I} \cup \mathcal{E}^*\}$ by traversing the tree in a depth-first manner. The general schema of the metric index is given in Figure 2.

We introduce the term *last split point* of $x[y]$ to refer to the starting position of

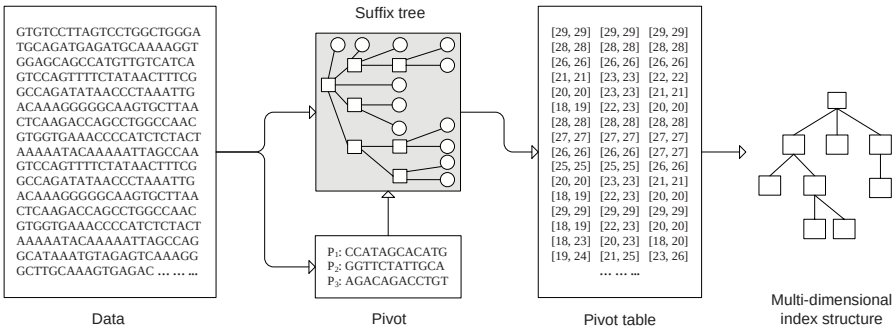


Figure 2: General schema of the metric index.

y in $x[y]$ (i.e., $|x|$). We now briefly describe the metric index.

First, a suffix tree $\mathcal{T}$ is built over S and a set of D pivots $\{P_1, \ldots, P_D\}$ selected from S. The pivot table is filled by columns as follows. For each pivot P_i, the algorithm traverses the suffix tree $\mathcal{T}$ in a depth first manner by filling the edit distance matrix row-wise and computing the *minimum* (*mind*) and *maximum* (*maxd*) distances between P_i and all the strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$. We note that *mind* and *maxd* distances for a string of $\{\mathcal{I} \cup \mathcal{E}^*\}$ are computed from the last split point of that string not from its beginning. For instance, the last split point of string "ssi[ssippi]" is 3 therefore we start considering *mind* and *maxd* from the position 3. Let N be an internal node of $\mathcal{T}$ and x be a substring of S which is obtained by visiting the nodes from the root to N by combining the labels of that path. Let us assume that the traversal algorithm has reached N. Then the overall costs for the computations of the edit distance between P_i and any child node of N can be reduced. Since they share same prefix (i.e., x) and the distance matrix values already filled up to N. Thus those distance values can be used for the distance computation for any child node of N.

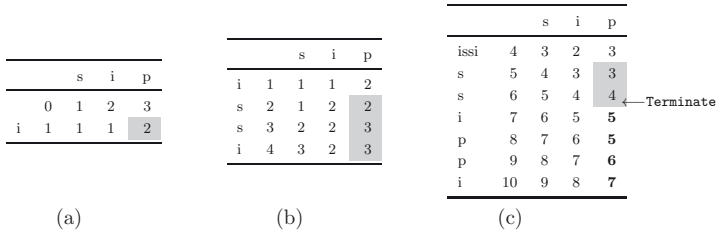
For each string of $\mathcal{I}$, we compute *mind* and *maxd* while for each string $x[y]$ of $\mathcal{E}$, we compute only *mind* and set *maxd* as $max(|x[y]|, |P_i|)$ because y can be very long. Let us assume that we have processed the string $xy_1 \ldots y_j$ of xy by using a matrix $ed_{0\ldots|xy_1\ldots y_j|,0\ldots|P_i|}$ and let v_j be $min_{1\leq j'\leq j} ed_{|x|+j',P_i}$. The distance computation for *mind* is early terminated at row j'' ($j < j'' < n$) if the following condition holds:

$$|x| + j'' - |P_i| > v_j \tag{1}$$

This is because $d(xy_1 \ldots y_{j''}, P_i) \geq |x|+j''-|P_i| > v_j$. Let us consider an example of computing *mind* and *maxd* between the pivot "sip" and strings "[i]", "i[ssi]" and "issi[ssippi]". The example is illustrated in Figure 3. For "issi[ssippi]", we stop the process after the sixth row due to the early termination condition (1). Thus *mind* and *maxd* between the pivot and strings "[i]", "i[ssi]", "issi[ssippi]" are $[2, 2]$, $[2, 3]$ and $[3, 10]$, respectively.

After the traversal of each pivot in P, we have a D-dimensional hyperrectangle with coordinates $[[mind_{x[y],P_1}, maxd_{x[y],P_1}], \ldots, [mind_{x[y],P_D}, maxd_{x[y],P_D}]]$ for each string $x[y]$ in $\{\mathcal{I} \cup \mathcal{E}^*\}$. Once we obtain the pivot table we can use any multidimensional index structure such as R-trees [4] to index it.

A query q with a radius r is resolved as follows. First, we compute



(a) (b) (c)

Figure 3: Examples of computations of the edit distance between pivot "sip" and strings "[i]" (a), "i[ssi]" (b) and "issi[ssippi]" (c). The values for *mind* and *maxd* are taken from the numbers in gray area. In the last figure, the last four numbers in bold listed after the termination point are not neccessary to be computed due to the condition 1.

distances between P and q to obtain a D-dimensional vector with coordinate $[d(q, P_1), \ldots, d(q, P_D)]$. With this vector, we define a query hyperrectangle with coordinates $[[l_1, h_1], \ldots, [l_D, h_D]]$, where $l_i = d(q, P_i) - r$ and $h_i = d(q, P_i) + r$ $(1 \leq i \leq D)$. The filtration of a string $x[y]$ can be done by using any pivot P_i $(1 \leq i \leq D)$ if at least one of $l_i > maxd_{x[y],p_i}$ or $h_i < mind_{x[y],p_i}$ holds, because $d(q, xy') > r$ then holds for any $xy' \in x[y]$. Informally, a candidate set consists of all the objects of $\{\mathcal{I} \cup \mathcal{E}^*\}$ whose hyperrectangles intersect the query hyperrectangle. In the refinement step, the candidate objects obtained from the previous step are checked against q with the suffix tree. For each candidate object $x[y]$, we may compute the edit distance between q and every prefix of x several times. In order to avoid this redundant work, we first mark every node in the path that represents every candidate object. After that, we traverse the suffix tree again, computing the edit distance between q and strings of those marked nodes of the suffix tree in the same way that we used in the pivot table construction. If $x[y]$ is part of the result set for q (i.e., $d(q, x[y]) \leq r$), we report all the starting points in the leaves of the subtree rooted by the node that represents $x[y]$.

4 Our algorithm

In the construction phase of the metric index, the replacement of the suffix tree with a suffix array leads to a new problem: finding all the substrings of explicit internal nodes of the suffix tree in the suffix array. Because suffix arrays do not have any hierarchical information that suffix trees have, these substrings are not readily available. However, by using some auxiliary information, we can deal with this problem without increasing the asymptotic running time. Our algorithm is based on the following two simple observations. First, all the paths through a node will be adjacent in the suffix array. Thus all split points (i.e., all nodes of $\mathcal{I}$) can be detected by examining all the adjacent pairs of the suffix array by analyzing their lcp values. For instance, the string "i[ssi]" of $i(2)$ is detected with the strings "issi[ppi]" of $e(3)$ and "issi[ssippi]" of $e(4)$. The corresponding lcp value of $e(4)$ in the lcp array is 4, therefore we selected the prefix "issi" of $e(3)$. Second, even with suffix trees we still need $\mathcal{O}(max(lcp) \cdot max(|P_i|))$ additional space in the worst case for a matrix that is used during computations of the edit distance between $\{\mathcal{I} \cup \mathcal{E}^*\}$ and P, because the distance values computed earlier are required in backtracking and

visiting other nodes of the suffix tree. Thus the *mind* and *maxd* of explicit internal node can be obtained from the matrix when the traversal algorithm is visiting a node or backtracking.

Algorithm 1 outlines the simplest version of our algorithm that generates all the equivalent strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$ of $\mathcal{T}$ for S. There are at most $n-1$ explicit internal nodes in $\mathcal{T}$. Thus, the algorithm generates at most $\mathcal{O}(2n)$ non-empty substrings of S. In the pseudocode, we use the standard stack operations such as *push*() (which adds a new element to the top of the stack), *pop*() (which removes an element from the top of the stack) and *top*() (which returns an element at the top of the stack). First, a stack *seen* is initialized in line 1. For any iteration i of the for-loop, any element e of *seen* represents that we have seen the internal node represented by the prefix of length e of $S[sa[i]]$. In line 6, the condition $seen.top() > next$ means that the algorithm needs backtracking. Thus, we need to pop out those positions greater than *next* from *seen*, because we check a different prefix with same length in the next time. However, the top element of *seen* that equals to *next* is not popped out, because that information is used in line 7 to check whether a node has been reported before. Also, the condition $next \neq 0$ ensures that we are not back to the root. The strings for $\mathcal{I}$ are reported in line 9. In line 10, each element of the suffix array is reported as an external node of $\mathcal{E}^*$, unless it has already been reported as explicit internal. Those nodes would be adjacent, so we only need to check the lengths of their strings.

Algorithm 1: Generate all the strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$

```
 1: initialize a stack seen
 2: for i = 1 to n do
 3:     cur ← lcp[i]; next ← 0
 4:     if i + 1 ≤ n then
 5:         next ← lcp[i + 1]
 6:         while seen ≠ ∅ and seen.top() > next do seen.pop()
 7:         if next ≠ 0 and (seen = ∅ or seen.top() < next) then
 8:             seen.push(next)
 9:             report S[sa[i], next]
10:     if next ≠ n − sa[i] then
11:         report S[sa[i]]
```

Algorithm 1 has a running time complexity of $\mathcal{O}(n)$. (The running time of the algorithm is equal to the number of times *seen.pop*() in line 6 is executed, and in line 8, *seen.push*() is executed at most $n-1$ times.)

We need two auxiliary arrays for filling up the pivot table columns with *mind* and *maxd*. The last split point of any string of $\mathcal{E}^*$ is defined by the maximum values of the current and next lcp while the last split point (*sp*) of any string of $\mathcal{I}$ is defined as follows. We can not direclty assign the lcp values to *sp* while detecting the nodes, because we may not obtain explicit internal nodes in depth-first traversal order of suffix tree nodes (for example, in our example $i(4)$ is reported after $i(5)$). Thus we maintain a stack that stores pairs of *sp* indexes and lcp values. The values of *sp* are assigned when we backtrack and select as the maximum value of the top element's lcp and the next lcp. The suffix array index *si* for an explicit internal node is an array of integers in which an element $si[i]$ indicates a smallest index of the suffix array where the string of the current explicit internal node is identical to the prefix

of $S[sa[i]]$ $(0 \leq i \leq n)$. This array is used during query processing. Now we modify algorithm 1 to generate the arrays of *sp* (Algorithm 2) and *si* (Algorithm 3) and to construct the pivot table (Algorithm 4). In Algorithm 4, the variable *row* points the last row in the matrix *ed* that has processed. We report *mind* and *maxd* and the suffix array indexes for each string in $\{\mathcal{I} \cup \mathcal{E}^*\}$.[1]

Sample runs of Algorithm 1, 2, 3 for string "mississippi$" are illustrated in Figure 4. Algorithm 1 produces the strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$ exactly in the same order that is shown the figure.

i	$sa[i]$	$lcp[i]$	$\mathcal{E}^*$
0	11	0	
1	10	0	
2	7	1	i[ppi]
3	4	1	issi[ppi]
4	1	4	issi[ssippi]
5	0	0	[mississippi]
6	9	0	p[i]
7	8	1	p[pi]
8	6	0	si[ppi]
9	3	2	si[ssippi]
10	5	1	ssi[ppi]
11	2	3	ssi[ssippi]

(a)

internal node	$\mathcal{I}$	$sp[i]$	$si[i]$
$i(1)$	[i]	0	1
$i(2)$	i[ssi]	1	3
$i(3)$	[p]	0	6
$i(5)$	s[i]	1	8
$i(4)$	[s]	0	8
$i(6)$	s[si]	1	10

(b)

Figure 4: Example runs of the algorithms for string "mississippi$". The strings of $\mathcal{E}^*$ (a) and the strings of $\mathcal{I}$ and the split points and the suffix array indexes for the strings of $\mathcal{I}$ (b).

The main principle of query processing for our approach is almost the same as in the original version. The only difference is that we mark the indexes of the suffix array instead of suffix tree nodes. Let us assume that we have processed a row k of the matrix for a candidate object. If the candidate is the result of a query at this time, the candidate is reported and we directly report those objects after the candidate object in the suffix array in contiguous order such that their lcp values are greater than or equal to k.

5 Experiments

In this section we provide experimental results. We are particularly interested in answering the following questions:

- How expensive is our simulation algorithm in terms of running time and memory requirement?
- What is the effect of varying the number of pivots?
- How does the performance of the metric index with *mind* and *maxd* compare to the one with only *mind*?
- How does the metric index work in practice?
- What are the main problem with the metric index, if any?

[1] In the pseudocode, the expression ($cond$? $expr_1$: $expr_2$) evaluates to $expr_1$ if $cond$ is true, and to $expr_2$, otherwise. Also, we let a_i denote an element at position $|a| - 1 - i$ of the array a $(0 \leq i < |a|)$.

Experiment Settings

First, we implemented our algorithm with full of indexing and query processing. For the original suffix tree based version, we implemented only the pivot table generation part (i.e., not query processing). We used two real-world datasets, namely DNA and protein datasets, which were obtained from the Pizza & Chilli corpus [2] and used a 10 MiB prefix of the datasets. The total number of objects (we recall that the size of $\{\mathcal{I} \cup \mathcal{E}^*\}$ is less than $2n$) to be indexed was 17 508 956 for DNA while for the protein data set the number was 17 341 406. The length of pivot and query was 35. Our query set consisted of 1000 queries, which were selected randomly from the datasets. For each query, we intentionally introduced 0–3 errors at random. We performed range searches with query radii varying from 0 to 3. The results were averaged over 10 runs. We did not use any compression algorithm during the construction of the index. For indexing the hyperrectangles, we used the R-tree implementation from the spatial index library [5][2]. All the programs were compiled by gcc 4.6.2 with the -O3 option and were run on a PC with a 3.3 GHz Intel Core i5-2500 processor and 8 GiB RAM. After testing several pivot selection algorithms, where none was clearly better than the others, we decided to use pivots that were randomly selected from the datasets.

Filtering effect of *maxd* and of number of pivot

Let MetricD be the metric index with only *mind* and MetricDD, the metric index with *mind* and *maxd*. The numbers of pivots used were 10 (p=10) and 20 (p=20). Figure 5 shows that the filtering effect of varying the number of pivot and MetricD versus MetricDD on various datasets.

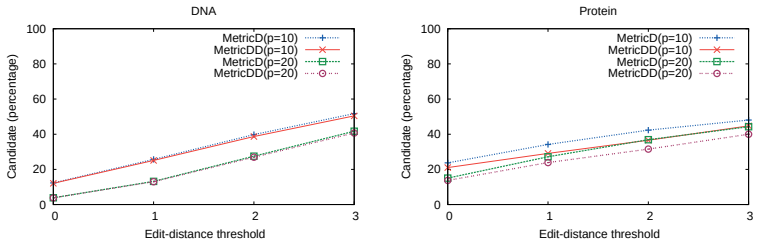


Figure 5: The percentage of candidates left after pivot filtration.

The experiments show that MetricDD filters out slightly more of the unpromising objects than MetricD. However, the difference is not really significant. It is also worth mentioning that MetricDD requires twice as much memory as MetricD.

Comparison of our simulation algorithm with the original one

We compared our simulation algorithm against the original one in terms of total memory and time required to generate pivot table. By total memory we mean the memory required to build a suffix tree for the original version while for our algorithm, we mean the sum of *sa*, *sp*, *si* and stacks and arrays that were used during the simulation. The comparison is shown in Table 1. The auxiliary arrays *sp* and *si* were generated in 0.19 s for both of the datasets.

[2]This implementation is commonly used in many papers of the database community

	Our simulation algorithm			The original algorithm		
Dataset	Total Mem. (MiB)	p=10 Time (s)	p=20 Time (s)	Total Mem. (MiB)	p=10 Time (s)	p=20 Time (s)
DNA	169	376	755	551	394	787
Protein	169	935	1844	545	937	1872

Table 1: Comparison of total memory and pivot table construction time.

Table 1 shows that our algorithm performs better than the original version in all of the experiments. We note that auxiliary arrays *sp* and *si* are not needed anymore after the pivot table construction.

Query response time

To decide on a multidimensional index structure to use in our experiments, we compared the performance of linear scan and R-tree on a pivot table with *mind* and *maxd* which was generated for 10 pivots. The R-tree was built over the pivot table. We set the query radius to 0 and measured the total time to answer a query set and required disk space. The results are shown in Table 2.

	Linear scan		R-tree	
Dataset	Time (s)	Disk space (MiB)	Time (s)	Disk space (MiB)
DNA	970	1967	4799	5291
Protein	1368	2058	5681	5250

Table 2: Comparison of linear scan and R-tree on pivot table.

Table 2 shows that the linear scan outperforms the R-tree on the pivot table. In light of this, the linear scan is used as the multidimensional index part of the metric index.

As a baseline competitor for the metric index, we used an enhanced linear scan. We performed a linear scan using the suffix and longest common prefix arrays to speed up the computations of the edit distance between query and suffixes. Figure 6 shows the comparison of query execution time for both indexes.

The experiments show that the metric index was beaten the enhanced linear scan by several order of magnitude. The reason for this bad performance of the metric index is due to the filtration step (we discussed about the space transformation in the second paragraph of Section 3). We divided query set execution time for each of filtration and refine steps. Then we converted them in the scale of 100 % and the results are presented in Figure 7.

The figures show that the filtration step took most of the query execution time. Because of this filtration effect, we did not compare our method to the original one in terms of search time. The distance distribution histograms for *mind* of both pivot tables are shown in Figure 8. We see that the distances are highly concentrated between 13 to 23 for DNA and 22 to 27 for protein data set.

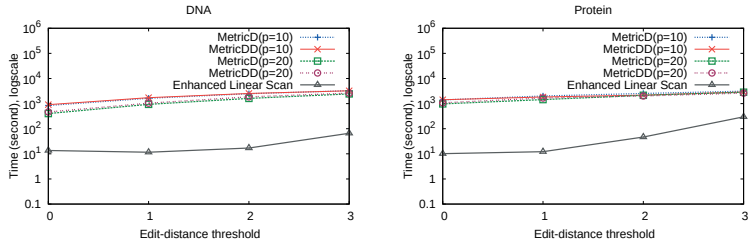


Figure 6: Comparison of query set execution time between the metric index and the enhanced linear scan.

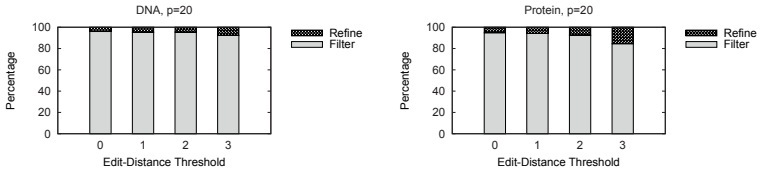


Figure 7: The percentage of filtration and refine steps in the query set execution time.

The histograms show that all the hyperrectangles more or less coincide. The *mind*s of the query object are also in this highly concentrated region. Because of this issue, the query hyperrectangle intersects with almost every indexed hyperrectangle. Thus the multidimensional search algorithm fails.

6 Conclusions

The primary goal of this paper was to empirically evaluate the metric index for approximate string matching based on suffix trees, introduced by Navarro et al. [9], as their original paper contained only theoretical results. In order to give the method a fair chance, we have proposed improvements that would reduce its memory consumption. We achieved this by simulating the index construction process using suffix arrays. Even with this improvement, though, our experiments on two real-data sets show that the metric index is impractical for real-world use, as it was clearly beaten by an enhanced version of a linear scan.

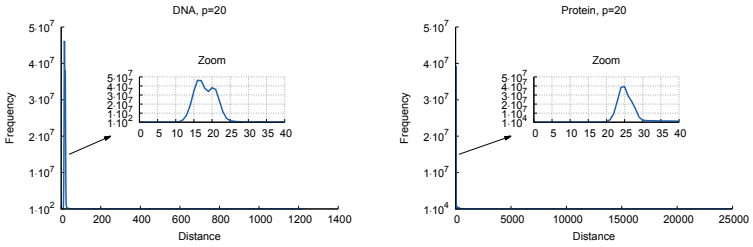


Figure 8: Distance distribution histograms of *mind* and its zoomed areas covering the distance values between 0 to 40 on X-axis.

Acknowledgements

We wish to thank Pål Sætrom for helpful discussions.

Appendix

Algorithm 2: Generate all the last split points for the strings of $\mathcal{I}$

initialize a stack *seen*
initialize a stack s that can store a pair of (lcp, pos)
for $i = 1$ **to** $n - 1$ **do**
  $cur \leftarrow lcp[i]; next \leftarrow lcp[i + 1]$
  while $s \neq \emptyset$ **and** $s.top().lcp > next$ **do**
    $c = s.pop().pos$
    $sp[c] \leftarrow s = \emptyset \ ?\ next : \max(next, s.top().lcp)$
  while $seen \neq \emptyset$ **and** $seen.top() > next$ **do** $seen.pop()$
  if $next \neq 0$ **and** $(seen = \emptyset$ **or** $seen.top() < next)$ **then**
    $seen.push(next)$
    $s.push(next, i)$
while $s \neq \emptyset$ **do**
  $c = s.pop().pos$
  $sp[c] \leftarrow (s = \emptyset)\ ?\ 0 : s.top().lcp$

Algorithm 3: Generate all the suffix array indexes for the strings of $\mathcal{I}$

initialize a stack *seen*
initialize an array a that can store a pair of (lcp, idx)
for $i = 1$ **to** $n - 1$ **do**
  $cur \leftarrow lcp[i]; next \leftarrow lcp[i + 1]$
  if $|a| = 0$ **then**
    $a.PushBack((next, i))$
  else if $a_{-1}.lcp = 0$ **then**
    $a_{-1} \leftarrow (next, i)$
  else if $a_{-1}.lcp < next$ **then**
    $a.PushBack((next, i))$
  else
    $a_{-1}.lcp \leftarrow next$
    while $|a| \geq 2$ **and** $a_{-2}.lcp > a_{-1}.lcp$ **do**
      $a_{-2}.lcp \leftarrow a_{-1}.lcp$
      $a.PopBack()$
  while $seen \neq \emptyset$ **and** $seen.top() > next$ **do** $seen.pop()$
  if $next \neq 0$ **and** $(seen = \emptyset$ **or** $seen.top() < next)$ **then**
    $seen.push(next)$
    $si[i] \leftarrow a_{-1}.idx$

Algorithm 4: Generate a pivot table column for a pivot p

initialize a stack *seen*
initialize a two dimensional array *ed*
$row \leftarrow 0$
for $i = 1$ **to** n **do**
  $cur \leftarrow lcp[i]; next \leftarrow 0$
  if $i + 1 \leq n$ **then**
    $next \leftarrow lcp[i + 1]$
    while $seen \neq \emptyset$ **and** $seen.top() > next$ **do** $seen.pop()$
    if $next \neq 0$ **and** ($seen = \emptyset$ **or** $seen.top() < next$) **then**
      $seen.push(next)$
      if $row + 1 \leq next$ **then**
        compute $d(S[row + 1 \ldots next], p)$
        $row \leftarrow next$
      $(mind, maxd) \leftarrow (min, max)\{ed[sp[i] + 1 \ldots next][|p|]\}$
      report $mind, maxd, si[i]$
  if $next \neq n - sa[i]$ **then**
    compute $d(S[row + 1 \ldots n - sa[i]], p)$ and let us assume that we are processing a row k ($row + 1 \leq k \leq n - sa[i]$) and then $mind \leftarrow min\{ed[row + 1 \ldots k][|p|]\}$ and early terminate the distance computation if the condition 1 holds
    $row \leftarrow$ (early terminated) ? $k : n - sa[i]$
    $row \leftarrow min(row, next)$
    report $mind$, $max(n - sa[i], |p|), i$

References

[1] L. Devroye, W. Szpankowski, and B. Rais. A note on the height of suffix trees. *SIAM J. Comput.*, 21(1):48–53, 1992.

[2] P. Ferragina and G. Navarro. Pizza & chili corpus. DNA and protein datasets downloaded on December 5th, 2011 from `http://pizzachili.dcc.uchile.cl`.

[3] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, pages 47–57, 1984.

[5] M. Hadjieleftheriou. Spatial index library v1.6.1. `http://research.att.com/~marioh/spatialindex`.

[6] S. Kurtz. Reducing the space requirement of suffix trees. *SoftwarePractice & Experience*, 29:1149–1171, 1999.

[7] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[8] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33, 1999.

[9] G. Navarro and E. Chávez. A metric index for approximate string matching. *Theoretical Computer Science*, 352:266–279, March 2006.

[10] M. Ruffalo, T. LaFramboise, and M. Koyutrk. Comparative analysis of algorithms for next-generation sequencing read alignment. *Bioinformatics*, 27(20):2790–2796, 2011.