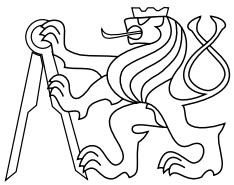
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

RESEARCH REPORT

ISSN 1213-2365

# Approximate Best Bin First $k$-d Tree All Nearest Neighbor Search with Incremental Updates

Jan Kybic, Ivan Vnučko

kybic@cmp.felk.cvut.cz

K333–40/10, CTU–CMP–2010–10

July 27, 2010

Available at
ftp://cmp.felk.cvut.cz/pub/cmp/articles/kybic/Kybic-TR-2010-10.pdf

This work was supported by by the Czech Ministry of Education project 1M0567.

**Research Reports of CMP, Czech Technical University in Prague, No. 10, 2010**

Published by

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

# Approximate Best Bin First $k$-d Tree All Nearest Neighbor Search with Incremental Updates

Jan Kybic, Ivan Vnučko

July 27, 2010

**Abstract**

We describe an approximate algorithm to find all nearest neighbors (NN) for a set of points in moderate to high-dimensional spaces. Although the method is generally applicable, it is tailored to our main application, which is a NN-based entropy estimation for an image similarity criterion for image registration. Our algorithm is unique for having simultaneously the following features: (i) It is usable for millions of data points in several tens of dimensions. (ii) It can deal with multiple points. (iii) It offers a speedup of the all-NN search task with respect to repeating a NN search for each query point. (iv) It allows exact as well as approximate search when reduced search time is needed. (v) The search tree can be updated incrementally when the change of values of the data points is small. The method is based on creating a balanced $k$-d tree, which is then searched using the best-bin-first strategy. The tree nodes contain both tight and loose bounding boxes. The method is presented using NN defined in an $\ell_\infty$ norm but can be applied to the $\ell_2$ norm, too.

We present a number of experiments demonstrating the behavior of our algorithm. We compare the presented method with a state-of-the-art approximate nearest neighbor search library ANN and we observe that the new method is superior for exact search and high number of points as well as for approximate search in small to moderate dimensions or when a fast approximation is needed.

# 1 Introduction

Nearest neighbor (NN) search problem consist of preprocessing a set of points $S$ such that for a given query point, a nearest neighbor from $S$ can be found quickly. This problem has a large number of applications, such as knowledge discovery in databases [1], case-based reasoning systems [2], computer aided medical diagnosis [3], protein or sequence search in molecular biology [4], scattered data interpolation [5], building a NN classifier (or a $k$-NN classifier) in pattern recognition [6, 7], or a vector quantization encoder in signal processing [8, 9]. It is also called a post-office problem [10]. Some of the many existing algorithms will be mentioned in Section 1.2. The task is most often formulated in a vector space using standard $p$-norms (Euclidean $\ell_2$ or Chebyshev $\ell_\infty$), although more general metrics such as the Levenstein (edit) distance for strings is used for some applications.

All nearest neighbor (all-NN) search is a less studied problem. It consists of finding a nearest neighbor from the set $S$ for all query points from $S$ [11, 12]. The task can be solved by repeating a standard NN search $n$ times, where $n$ is the number of points in $S$, but there are computational savings to be made given that the query set is identical to the set of points $S$ and both are known in advance. All-NN search is useful in estimating a local probability density from a set of samples, based on the observation that small NN distances correspond to points with high probability density and vice versa [13]. Applications include manifold learning [14], estimation of local dimension [15], estimation of Rényi entropy [16], Shannon entropy [17, 18, 19], Kullback-Leibler divergence [20], and mutual information [21].

Our target application is estimating mutual information based similarity criteria for image registration. Image registration or image alignment is one of the fundamental image analysis tasks, especially in the biomedical domain. Many image registration methods are based on optimizing an image similarity criterion and mutual information (MI) is a frequently used image similarity criterion, especially in multimodal image registration [22, 23, 24, 25], for its ability to explore the statistical dependency between images (see also Section 1.4). Normally, mostly scalar pixel intensities are used as features, even though some attempts to use higher dimensionality vector features have been made [26, 27, 28]. The main obstacle is that the predominantly used standard histogram-based estimator performs poorly in higher dimensions. Fortunately, entropy estimators usable in higher dimensions exist. In particular, the Kozachenko-Leonenko (KL) estimator [18, 19] is based on NN distances. We have shown earlier [29] that the KL estimator can be successfully used to evaluate MI image similarity criteria in up to 25 dimensions, which requires evaluating entropy in dimension $d = 50$.

The computational bottleneck of the KL estimator is the all-NN search, since the number of pixels $n$ in an image is large. In this article we describe a practical all-NN search algorithm that can be used together with the KL entropy estimator for image similarity evaluation in image registration. This application has a number of specific requirements which we believe cannot be found simultaneously in any existing method — this justifies developing a specialized algorithm:

(i) The datasets are large and the all-NN search is run once for each iteration of the optimizer. Therefore speed is very important. The number of data points $n$ (corresponding to the number of pixels) is typically between $10^5$ and $10^6$ and can be up to $10^7$. The typical number of dimensions $d$ is between 4 and 50.

(ii) Due to quantization, the dataset can contain the same point several or many times. This can be for example caused by a homogeneous background in the images. The all-NN search algorithm must handle these cases efficiently and report the data point multiplicities.

(iii) The algorithm can take advantage of the fact that the query set is identical to the set of data points and both are known in advance.

(iv) As exact all-NN search is likely to be too slow for moderate to large $n$ and $d$, we also need the ability to calculate an approximate solution for a given time budget.

(v) The data points depend continuously on a geometrical transformation being controlled by an optimizer. We can therefore assume that especially at later stages of the optimization, the changes of data point values between iterations will be small. Our all-NN search algorithm can take advantage of this fact.

The rest of this paper is organized as follows: After defining the problem in Section 1.1 and reviewing some of the existing algorithms in Section 1.2, we describe our target application in Section 1.3. Our proposed all-NN search algorithm is described in Section 2. We finish with an experimental evaluation of our C++ implementation in Section 3.

## 1.1 Problem definition and notation

Given a set of $n$ points $S$ from a vector space $V = \mathbb{R}^d$ equipped with a norm $\|\cdot\|$, and a query point $\mathbf{q} \in V$, than $\mathbf{p} \in S$ is a *nearest neighbor* (NN) of $\mathbf{q}$ iff

$$\varrho_{\mathbf{q}} = \|\mathbf{p} - \mathbf{q}\| = \min\bigl\{\|\mathbf{p}' - \mathbf{q}\|; \mathbf{p}' \in S\bigr\} \tag{1}$$

and $\varrho_{\mathbf{q}}$ is the NN distance. We will write $\mathbf{p} = \mathrm{NN}_S(\mathbf{q})$. Note however that a NN may not be unique and that the NN relation is not symmetric.

It is assumed that many queries will be performed for the same $S$. Therefore, we should preprocess $S$ first, so that subsequent queries can be answered fast (in sublinear time).

Since exact NN search is often too time consuming for many application, especially for high dimensional spaces (high $d$) and many points (high $n$), an *approximate NN* (aNN) problem is sometimes considered. It does not guarantee to find a true NN but attempts to find a point from $S$ which is "close enough". The closeness can be specified in terms of a distance, e.g. searching for a point in a distance at most $(1+\epsilon)\varrho_{\mathbf{q}}$, where $\varrho_{\mathbf{q}}$ is the true NN distance [30]. We relax the requirements here, judging the approximation quality by an application dependent error, in our case the entropy estimation error (see Section 1.3).

*All NN problem* (all-NN) consists of finding, for all query points $\mathbf{q}$ from $S$, a NN from $\tilde{S} = S \backslash \{\mathbf{q}\}$ [11, 12]. Our generalization of the formulation is to consider $S$ to be a *multiset* [31, 32], i.e. it can contain some points several times, as this occurs often in our data. We write $\chi_S(\mathbf{p}) = k$ to indicate that $S$ contains $k$ times the point $\mathbf{p}$, where $\chi_S$ is a generalized indicator function and we interpret $S = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$ as $S = \bigcup_{i=1}^{n} \{\mathbf{p}_i\}$. We denote $S^e = \{\mathbf{p} \in V; \chi_S(\mathbf{p}) > 0\}$ the underlying set of elements of the multiset $S$. Then the all-NN problem can be defined as follows: Given a multiset $S$ with elements $S^e \subset V = \mathbb{R}^d$, find a mapping $\mathrm{NN}' : S^e \to S^e$ such that

$$\mathrm{NN}'_S(\mathbf{q}) = \mathrm{NN}_{\tilde{S}^e}(\mathbf{q}) \quad \text{for all } \mathbf{q} \in S^e \tag{2}$$

where $\tilde{S}^e$ is the underlying set of elements of $\tilde{S} = S \backslash \{\mathbf{q}\}$, with the nearest neighbor distance $\varrho_{\mathbf{q}} = \|\mathbf{q} - \mathrm{NN}'_S(\mathbf{q})\|$. In other words, if the query point $\mathbf{q}$ has multiplicity one, as in standard sets, then $\mathrm{NN}'$ returns its nearest neighbor from the rest of the points, as defined previously. Otherwise, if the query point has multiplicity $\chi_S(\mathbf{q})$ greater than one (which is only allowed in multisets) we return $\mathbf{q}$ itself, the nearest neighbor distance is zero, $\varrho_{\mathbf{q}} = d(\mathbf{q}, \mathbf{q}) = 0$, and we will also report the multiplicity $\chi_S(\mathbf{q})$.

This definition is motivated by our entropy estimation application (Section 1.3). As in the single NN case, there may be several possible NN points for some $\mathbf{q}$; we allow $\mathrm{NN}'_S(\mathbf{q})$ to return any of them. This is not a problem because as we will see later, in our application we are only interested in the NN distance $\varrho_{\mathbf{q}}$.

The *dynamic all-NN problem* consists of solving the all-NN problem for $m$ sets of points $S_i = \{\mathbf{p}_1^{(i)}, \mathbf{p}_2^{(i)}, \ldots, \mathbf{p}_n^{(i)}\}$, with $i = 1, \ldots, m$, each of them containing $n$ points, assuming that the displacement $\|\mathbf{p}_j^{(i+1)} - \mathbf{p}_j^{(i)}\|$ of each

point $j$ is small with respect to the inter-point distances $\|\mathbf{p}(i)_j - \mathbf{p}(i)_k\|$. In other words, $\mathbf{p}_j^{(i)}$ is seen as a trajectory of a point $\mathbf{p}_j$ in time $i$ and it is assumed that the movement is slow. This allows the datastructure built by preprocessing the set $S_i$ to be used with few changes also for the set $S_{i+1}$, bringing computational savings. As far as we know, this scenario has not yet been described in the literature.

In the sequel, we will work with the $\ell_\infty$ (maximum or Chebyshev) norm

$$\|\mathbf{x} - \mathbf{y}\| = \|\mathbf{x} - \mathbf{y}\|_\infty = \max_i |x_i - y_i|$$

where $x_i$ and $y_i$ are the components of $\mathbf{x}$ and $\mathbf{y}$, respectively. As most NN search algorithms (including our one, described in Section 2) are based on partitioning the space into axis-parallel rectangles, this choice allows them to perform slightly better than with the Euclidean norm $\ell_2$ which leads the spherical neighborhoods. However, for most data, the performance is rather similar. There are only minor changes required to adapt our algorithm to other $\ell_p$ norms, namely $\ell_2$, thanks to the discrete norm equivalence relations such as

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{d}\|\mathbf{x}\|_\infty \tag{3}$$

which enable us to convert the distance bounds between norms.

This article will describe an algorithm for exact and approximate all nearest neighbor search in moderate to high number of dimensions and for a large number of points.

## 1.2 Existing NN search algorithms

Nearest neighbor search in low dimensions is a well-studied problem. For $d = 1$, an asymptotically optimal solution consist of sorting the points and applying a binary search. For $d = 2$, an asymptotically optimal solution involves a Voronoi diagram [33, 30], however, it is slow in practice. A number of algorithms is based on griding or treating each dimension separately, thus trimming the set of candidate points and exhaustively searching the remaining ones [34, 35, 36]. Unfortunately, these strategies only seem to be suitable for low dimensions and distributions close to uniform.

An often used technique for moderate $d$ ($2 \sim 100$) and large $n$ is hierarchical partitioning of the space or the data points [30, 33, 37], creating a tree structure. Quad-tree methods [38, 39] divide the space to $2^d$ equal hyperrectangles at each level; hyperspheres are also used [40, 41]. A prominent example of a data-partitioning approach is the $k$-d tree [42, 43, 44, 45, 46], based on splitting the data points evenly between node children at each

level by a an axis parallel dividing plane. We are using an augmented $k$-d tree in the proposed algorithm (Section 2.2). A balanced box-decomposition (BBD) tree guarantees that both the cardinality and size of the partitions decreases exponentially [30], avoiding degenerate splits. A well-separated pair decomposition [47] constructs a list of point decompositions, speeding up the $k$-nearest neighbor search.

For large $d$ and $n$, solving the exact NN problem is impractical, as it is very likely going to be too slow for many intended applications. In fact, the complexity of most exact NN methods is exponential in $d$ and there is no known method which is simultaneously approximately linear in storage space and logarithmic in query time [30, 48, 33]. Approximate NN search can guarantee bounds on the error of the reported NN distance [30]. A spill tree [49] method traverses the tree without backtracking while making the boxes overlap, returning the correct NN point with high probability [49].

For very large number of dimensions ($d > 100$), the most promising approximative approaches are based on random projections [50] or locality sensitive hashing [51, 52, 53], especially if the intrinsic dimensionality of the data is large and a large number of errors is allowed [54], otherwise spatial tree based approaches are to be preferred [49].

Dynamic NN search algorithms address the case of solving the NN or all-NN problem repeatedly, with a changing point set $S$. However, as far as we know, all previously described methods handle only the inserting and deleting of points between queries [55, 56, 57, 58] or the case of points moving along a predetermined trajectories [45]. This is different from our problem, where the points move slowly but in an unpredictable way.

Finally, let us mention two classes of NN search algorithms which are outside the scope of this article: A metric tree [59] or a vantage point tree [60] can be used for a search in a metric space which is not a vector space. Finally, there are NN search algorithms for very large data sets kept in external memory, minimizing I/O operations.

## 1.3 Nearest neighbor entropy estimation

Our target application is estimating entropy using a little known Kozachenko-Leonenko (KL) NN-based entropy estimator [18, 19]. Given a set of $n$ samples $S$ of a random variable $\mathsf{F}$ in $\mathbb{R}^d$ with a probability density distribution $f$, then the Shannon entropy of $\mathsf{F}$

$$H(\mathsf{F}) = -\int_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) \log f(\mathbf{x}) \mathrm{d}\mathbf{x} \tag{4}$$

can be estimated as

$$\hat{H}(\mathsf{F}) = -\frac{1}{n}\Big(\sum_{\mathbf{q}\in S} d \log \varrho_{\mathbf{q}}\Big) + \gamma + \log 2^n (N-1) \tag{5}$$

where $n$ is the number of samples, $d$ is the dimensionality of the space, $\varrho_{\mathbf{q}} = \|\mathbf{q} - \mathrm{NN}_{\tilde{S}}(\mathbf{q})\|_\infty$ is the $\ell_\infty$ NN distance for a point $\mathbf{q}$ in $\tilde{S} = S \setminus \{\mathbf{q}\}$ as defined by (1), and $\gamma \approx 0.577$ is the Euler constant. The original formulation uses an $\ell_2$ distance [18, 29]; here we present a version using an $\ell_\infty$ norm [61] for a better compatibility with the space partitioning strategy of NN algorithms, as mentioned above. However, the estimators (5) for $\ell_2$ and $\ell_\infty$ distances are almost identical, differing only in the constant term.

As this estimator does not use binning, it is more suitable for high dimensionality data and/or small sample sizes than the standard histogram approach, mainly because of its small bias. This can be shown experimentally [19, 29, 62]. The estimator (5) is asymptotically unbiased and consistent under very general conditions on the smoothness and integrability of the probability density function $f$. The mathematical details can be found in [18, 63], here we only note that in practice the estimator works as long as $f$ is bounded, e.g. Dirac impulses are not allowed. This in turn means that the probability of two points from $S$ coinciding and thus $\varrho_{\mathbf{q}} = 0$ in (5) is zero, so it appears that no precautions are needed to avoid taking the logarithm of zero in (5). However, in practice the data often come from finite accuracy measurements; in our target application of image registration, the image intensity values are often quantized into as few as 256 levels, and the conflicts (i.e. $\varrho_{\mathbf{q}} = 0$) happen frequently. Some authors advocate adding a low-amplitude perturbation to the data [21]. We prefer to robustify the estimator by modifying the $\log \varrho_{\mathbf{q}}$ function in (5). The simplest choice is a thresholded version $\log' \varrho_{\mathbf{q}}$ [61]

$$\log' \varrho_{\mathbf{q}} = \begin{cases} d \log \varrho_{\mathbf{q}} & \text{for } \varrho_{\mathbf{q}} \geq \varepsilon \\ \log\big(\varepsilon^d / \chi_S(\mathbf{q})\big) & \text{for } \varrho_{\mathbf{q}} < \varepsilon \end{cases} \tag{6}$$

where the threshold $\varepsilon$ corresponds to the measurement accuracy or a quantization bin size. This can be viewed as calculating an upper bound on the entropy or switching to a kernel plug-in estimator for $\varrho_{\mathbf{q}} < \varepsilon$. A more principled approach replaces the hard switching by a smooth transition obtained through numerical optimization [62]. In both cases, the all-NN search algorithm must be able to correctly work with data containing multiple points, hence our extended definition of the all-NN problem in Section 1.1.

### 1.4 Mutual information and image registration

The entropy estimate (5) can be used to calculate a mutual information (MI) between two random variables $\mathsf{X}$, $\mathsf{Y}$ [64]

$$I(\mathsf{X};\mathsf{Y}) = H(\mathsf{X}) + H(\mathsf{Y}) - H(\mathsf{X},\mathsf{Y})$$

where $H(\mathsf{X})$ and $H(\mathsf{Y})$ are marginal entropies and $H(\mathsf{X},\mathsf{Y})$ is the joint entropy. Mutual information has shown to be a very promising image similarity criterion especially for multimodal image registration [23], because it can capture very general and not necessarily functional dependences. Pixel intensity values from the two images being registered are typically considered as samples (data points) of the random variables $\mathsf{X}$, $\mathsf{Y}$.

Image registration [65] is an important step in many computer vision and medical imaging applications such as movement compensation, motion detection and measurement, image mosaicking, image fusion, atlas building and atlas based segmentation. So far, most registration algorithms based on a maximization of a MI criterion estimate the entropy (and hence MI) using a histogram-based plug-in estimator [66, 67] or a kernel density estimator [68], using scalar gray-scale image values as features, i.e. requiring entropy estimation for $d = 2$, for samples $\mathbf{x}_i = \begin{bmatrix} f_i & g_i \end{bmatrix}$ where $f_i$ and $g_i$ are pixel values in the two images being compared. There are only a few attempts in the literature to use more complicated, *high dimensional features*, such as gradients, color, output of spatial filters, texture descriptors, or intensities of neighborhood pixels [26, 27, 28, 69]. The main difficulty stems from the inappropriateness of a histogram estimator for high-dimensional data. For this reason, these above mentioned approaches are limited to low dimensions (e.g. $d = 4$ in [26]) or need to use the normal approximation ([28] with $d = 18$).

Using the KL entropy estimator (5) is a qualitative improvement [29, 61], permitting evaluation of the entropy for truly high-dimensional features such as color ($d = 6$), texture descriptors, or large pixel neighborhoods ($d = 10 \sim 100$).

## 2 Method

Our method uses a $k$-d tree structure [42]. Apart from using standard dividing planes, creating what we call loose bounding boxes (LBB), we maintain also a tight bounding box (TBB) structure (see Section 2.2) that permits a more efficient pruning during search. We use the best-bin-first (BBF) strategy [44]: we start to search the tree nodes closest to the query point (that

are most likely to contain the NN) and then expanding the search to nodes further away. When a predetermined computational budget is exhausted, we stop the search and declare the best point so far as the approximative NN. This way we can adjust the trade-off between accuracy and time complexity.

Another improvement concerns the all-NN search: We start at the leaf containing the query point, instead of at the tree root. By doing this, we save one tree traversal in comparison with the naive approach of repeating $n$ times the standard NN search,

Finally, we obtain additional computational savings by taking advantage of the assumed small change pattern which makes it possible to partly reuse a previously built $k$-d tree structure, making only local adjustments instead of rebuilding the tree completely.

An important practical consideration is that in our application identical data points can occur, as mentioned in Section 1.3. Our algorithm therefore keeps track of and reports point multiplicities.

## 2.1 Algorithm overview

The method consists of three related routines:

- The `BuildTree` routine (Section 2.3) takes a multiset $S$ of $n$ points from $\mathbb{R}^d$ and creates an extended $k$-d tree $T$. (Section 2.2).

- The `SearchTree` routine (Section 2.4) takes a multiset $S$ and the corresponding tree $T$ and calculates a mapping $\mathbf{q} \mapsto \big(\mathrm{NN}'_S(\mathbf{q}), \chi_S(\mathbf{q})\big)$ for each $\mathbf{q} \in S^e$ (see Section 1.1 to recall the notation). In other words, for each data point $\mathbf{q}$ that occurs in $S$ exactly once, the mapping returns its nearest neighbor among the rest of the points in $S$ and $\chi_S(\mathbf{q}) = 1$; for each point $\mathbf{q}$ that occurs several times ($\chi_S(\mathbf{q}) > 1$), the mapping returns $\mathbf{q}$ itself and its multiplicity $\chi_S(\mathbf{q})$.

- The `UpdateTree` routine (Section 2.5) takes a multiset $S = \big\{\mathbf{p}_1, \ldots, \mathbf{p}_n\big\}$ with the corresponding tree $T$. It further takes a modified multiset $S' = \big\{\mathbf{p}'_1, \ldots, \mathbf{p}'_n\big\}$, where $\mathbf{p}_i$ and $\mathbf{p}'_i$ are corresponding points from $S$ and $S'$, respectively. The `UpdateTree` routine modifies the tree $T$ so that it corresponds to $S'$ and can be used as if it were calculated directly from $S'$ by `BuildTree`. Using `UpdateTree` should be more efficient than creating a fresh tree by `BuildTree`, provided that the changes in point coordinates $\mathbf{p}'_i - \mathbf{p}_i$ are small.

## 2.2 Extended $k$-d tree

We use an extended version of the $k$-d tree data structure [42, 70, 71]. A $k$-d tree stores data points in its leaves. Each node represents a hyperinterval (an axis aligned hyperrectagle), which we shall call a *loose bounding box* (LBB) of the node. The LBB can be finite, infinite or semi-infinite. A LBB of a node contains all points in the subtree of this node. The LBBs are stored explicitly in each node and are constructed recursively as follows: The bounding box of the root node corresponds to the whole space, $\mathrm{LBB} = \mathbb{R}^D$. Then, each non-leaf node stores a splitting dimension $m$ and value $\xi$ that define an axis parallel hyperplane; the LBB of the left resp. right child is determined as

$$\begin{aligned} \mathrm{LBB_L} &= \left\{x_m < \xi; \mathbf{x} \in \mathrm{LBB_P}\right\} \\ \mathrm{LBB_R} &= \left\{x_m \geq \xi; \mathbf{x} \in \mathrm{LBB_P}\right\} \end{aligned} \tag{7}$$

where $\mathrm{LBB_P}$ is a LBB of the parent and $x_m$ is the $m$-th component of the vector $\mathbf{x}$.

Additionally, each leaf node contains a multiset of points from $S$ that belong to its LBB. The LBBs of the leaf-nodes (as well as the LBBs at all other levels of the tree) form a complete partitioning of the space $\mathbb{R}^D$, and hence all points from $S$ can be uniquely attributed to leaves. Note that duplicate points are guaranteed to fall into the same leaf.

As a novelty, we also maintain and store a tight bounding box (TBB) for each node. The TBB of a node $Q$ is the smallest hyperinterval (in the sense of inclusion) containing all points in the subtree of $Q$ and is given by their minimal and maximal coordinates

$$\begin{aligned} TBB_Q &= \prod_{i\in\{1\ldots d\}} [l_i^Q; h_i^Q] = \prod_{i\in\{1\ldots d\}} \left\{\mathbf{x}; \mathbf{x} \in \mathbb{R}^d, l_i^Q \leq x_i \leq h_i^Q\right\} \\ \text{with} \quad l_i^Q &= \min_{\mathbf{y}\in Q} y_i, \qquad h_i^Q = \max_{\mathbf{y}\in Q} y_i \end{aligned} \tag{8}$$

where $\mathbf{x} = (x_1, \ldots, x_d)$ are the point coordinates, $\Pi$ denotes a Cartesian product, and $\mathbf{y} \in Q$ are all data points from the subtree of $Q$. In Figure 1a, TBBs are drawn in dashed lines. The purpose of the TBBs is to improve the efficiency of pruning during the search, since a TBB is always smaller than the corresponding LBB.

For the purpose of the dynamic tree update procedure we introduce a parameter $\delta \in \langle 0, 0.5\rangle$, limiting each child subtree to contain at most $(\frac{1}{2} + \delta)n_p$ points, where $n_p$ is the number of points of its parent subtree. This creates a $\delta$-pseudo $k$-d tree [45], where the parameter $\delta$ controls the trade-off between

the update speed and the speed of search on the updated tree (Section 2.5). Note that $\delta = 0$ corresponds to a perfectly balanced tree, white $\delta = 1/2$ allows the tree to be completely degenerated to a linear list of nodes. For easier housekeeping, each node stores the number of points in its subtree.

## 2.3 Building the tree (`BuildTree`)

The tree building algorithm (see Algorithm 1) is standard, based on recursive splitting. We start with the entire input multiset $S$ as the root node and the whole space $R^D$ as its loose bounding box LBB. Then, recursively from the root, for each node $Q$ we choose the splitting dimension $m$ as the longest edge of the TBB

$$m = \arg \max_{i \in \{1 \dots d\}} (l_i^Q - h_i^Q)$$

where the TBB boundaries $l_i^Q, h_i^Q$ are defined in (8). The splitting value $\xi$ is the median of $\{x_m; \mathbf{x} \in Q\}$ [43]. In this way, the tree is balanced with respect to the number of points in each subtree. The subdivision is stopped when the number of points to process is less than or equal to a predetermined parameter $L$ (maximum number of points in a leaf). Then a TBB is computed for each newly created node according to (8) by reexamining all points in the node; the LBB is created in constant time using (7).

The asymptotically most time consuming step is calculating the TBB which takes $O(md)$ time per node, where $m$ is the number of points in a node; median calculation takes $O(m)$ time per node. The total asymptotic time complexity of building the tree is therefore $O\big(dn \log(n/L)\big)$, where $\log_2(n/L)$ is the depth of the tree.

Note that identical points are never separated by splitting and end up in the same leaf node. However, the consequence is that perfect balancing may not be achievable. If the splitting is completely degenerated (one of the children receives all the points), the node is split no further. Therefore the depth of the tree and hence the `BuildTree` complexity is not increased.

## 2.4 Exact nearest neighbor search (`SearchTree`)

We describe here an exact all-NN search algorithm. We loop through all leaves and through all points in each leaf. Each point $\mathbf{q}$ acts as a query points and we find its nearest neighbor $\hat{\mathbf{q}}$. The method is described in Algorithm 2, ignoring the parameter $V$ for the moment. The basis for the search is the *BBF (best bin first)* tree traversal [44], with pruning. Additionally, instead of starting from the global root, the search starts from the leaf $Q$ containing the query point $\mathbf{q}$. Node $Q$ can be temporarily regarded as a tree root which

**Algorithm 1**: `BuildTree`, for building the extended $k$-d tree.

**Input**: Multiset of points $S$ from $\mathbb{R}^d$.
**Output**: $k$-d tree $T$ for $S$.

**function** `BuildTree`($S$):
  create root node $R$ with points $S$ and $\mathrm{LBB}_R = \mathbb{R}^d$
  `SplitNode(`$R$`)`
**return** tree $T$ with root $R$

**function** `SplitNode`(*tree node $Q$ with points $P$ and $LBB_Q$*):
  compute $\mathrm{TBB}_Q$ as follows:
    $l_i^{\mathrm{Q}} = \min\{x_i; \mathbf{x} \in P\}$, $h_i^{\mathrm{Q}} = \max\{x_i; \mathbf{x} \in P\}$
  **if** number of points $\|P\| > L$ **then**
    $m \leftarrow \arg\max_{1 \le i \le D} \left(x_i^{\mathrm{H}} - x_i^{\mathrm{L}}\right)$
    $\xi \leftarrow$ `Median` $\{x_m;\ \mathbf{x} \in P\}$
    $P_L \leftarrow \{x_m < \xi; \mathbf{x} \in P\}$
    $P_R \leftarrow \{x_m \ge \xi; \mathbf{x} \in P\}$
    $\mathrm{LBB_L} \leftarrow \left\{x_m < \xi; \mathbf{x} \in \mathrm{LBB_Q}\right\}$
    $\mathrm{LBB_R} \leftarrow \left\{x_m \ge \xi; \mathbf{x} \in \mathrm{LBB_Q}\right\}$
    **if** neither $P_L$ nor $P_R$ is empty **then**
      create left child $L$ of $Q$ with points $P_L$ and $\mathrm{LBB}_L$
      create right child $R$ of $Q$ with points $P_R$ and $\mathrm{LBB}_R$
      `SplitNode(`$L$`)`
      `SplitNode(`$R$`)`

(temporarily) redefines the edge directions (see Figure 2). If $k > 1$ points identical to $\mathbf{q}$ are found in node $Q$ (including $\mathbf{q}$), we can stop the search and report the multiplicity $k$ of $\mathbf{q}$. Otherwise, we start to traverse the tree starting from $Q$.

While traversing the tree, the points in so far unseen leaf nodes are searched and the currently best NN candidate $\hat{\mathbf{q}}$ is updated. The search order is determined according to the best-bin-first strategy [44] using a lower bound $\eta_X$ of the distance from the query $\mathbf{q}$ to yet unexplored points reachable from a node $X$. The idea is that nodes close to $\mathbf{q}$ are more likely to contain the NN to $\mathbf{q}$. The lower bound $\eta_X$ is calculated as follows: If $X$ is not an ancestor of $Q$ (in the sense of the original edge orientations), then $\mathbf{q}$ lies outside of $X$ and $\eta_X$ is the distance of $\mathbf{q}$ to its TBB, $\eta_X = d\big(\mathbf{q}, \mathrm{TBB}_X\big)$. Otherwise, if $X$ is an ancestor of $Q$, then $\mathbf{q}$ lies inside $X$. As we have arrived to $X$ from $Q$, it means that $X$ must have a child $Z$ that we have already explored and that is equal to $Q$ or is an ancestor of $Q$ . We then set $\eta_X$ as the distance of $\mathbf{q}$ to the yet unseen part of the $X$ subtree, i.e. the complement of $\mathrm{LBB}_Z$, $\eta_X = d\big(\mathbf{q}, \mathbb{R}^D \setminus \mathrm{LBB}_Z\big)$. Both cases are illustrated in Figure 2a as $\eta_H$ and $\eta_A$, respectively.

When a node is visited, both nodes accessible from it are inserted into a priority queue, if they have not been seen before. The priority queue is sorted according to $\eta$. A new node to visit is taken from the top of the queue, with the smallest $\eta$. If the value $\eta$ of a node is higher than the distance to the currently best NN candidate $\hat{\mathbf{q}}$, then such a node is not inserted into the queue at all, because it cannot lead to a point closer than the currently best distance. If such a node is found at the top of the queue, than the search can be terminated.

To determine which nodes are unseen and thus what edges should be taken, our implementation stores in the priority queue not only the node to visit but also its predecessor (in the temporary orientation). Alternatively, it is possible to mark visited nodes by a special flag, but care must be taken to properly unmark them afterwards.

Most of the time, especially in low dimensions, the NN is found inside the leaf $Q$ and very few other leaves are searched. Then the total time for an all-NN search is $O(ndL)$ (see Section 3 for experimental results). As the intrinsic dimension of the dataset increases, the neighboring boxes have to be searched more and more often, making the search time increase exponentially with $d$. In the worst case, all points have to be searched, leading to a total time $O(n^2 d)$, the same as for the brute force algorithm.

### 2.4.1 The approximative search

The algorithm described in the previous section guarantees to find the exact NN but can be slow for many applications. Here we describe a simple change, leading to an approximative, but potentially much faster version. It is controlled by a parameter $V$ that bounds the number of visited points. If this number is exceeded, the search is stopped and the best result so far is reported. The parameter $V$ bounds the worst-case time complexity of the approximate all-NN search to $O(Vn)$. Algorithm 2 already incorporates this change. Alternatively, it is also possible to limit the CPU time per search.

## 2.5 Updating the tree (`UpdateTree`)

As mentioned in Sections 1 and 1.1, the changes between point positions in subsequent invocations of the all-NN search algorithm are often small in our application and updating the tree can therefore be faster than rebuilding it from scratch. The `UpdateTree` routine (Algorithm 3) takes a multiset $S$ and a corresponding tree $T$ and updates the tree to correspond to a new set of points $S'$ (see also Section 2.1). We assume that the number of points is the same and there is a one-to-one mapping between the points in $S$ and $S'$.

The `UpdateTree` (Algorithm 3) routine detects points from the tree which have moved out of the LBBs of their original leaves and attributes them to the appropriate new leaves. The TBBs of affected nodes are updated. Finally, parts of the tree violating the balance condition (Section 2.2) is are rebuilt.

The method consists of two depth-first recursive traversals of the tree. The first pass (procedure `UpdateNode`) is a bottom-up postfix traversal (parents are examined after their children). Each node is checked for points which lie outside of the node's LBB and such points are temporarily assigned to the node's parent. (Recall that normally only leaf nodes contain points.) As a result of the first pass, all points lie in the LBBs of their nodes.

The second pass (procedure `RebuildNode`) is a top-down prefix traversal (parents before children) and serves to assign points held by non-leaf nodes to the appropriate leaves. A point is moved to the left resp. to the right child according to the splitting dimension $m$ and the splitting value $\xi$ of the parent node (as in Algorithm 1). TBBs are updated in a postfix order (parent after children) during the same pass. If an unbalanced subtree is detected (Section 2.2), the subtree is rebuilt using the `SplitNode` procedure (Algorithm 1). The tolerated amount of unbalance is determined by a parameter $\delta$. See Section 3 for an experimental evaluation of its effect.

In the best case, the tree is traversed twice, the points are checked but no other work is done, with a complexity $O(dn)$. In the worst case, the

**Algorithm 2**: `allNNSearch` finds the mapping $\mathbf{q} \mapsto \big(\hat{\mathbf{q}}, \chi_S(\mathbf{q})\big)$ for each point $\mathbf{q}$ in the tree $T$. If $V = \infty$, then $\hat{\mathbf{q}} = \mathrm{NN}'_S(\mathbf{q})$ is the exact NN, otherwise $\hat{\mathbf{q}} \in S^e$ is an approximate NN.

**Input**: Tree $T$ with points $S$, maximum number of visited points $V$
**Output**: Exact or approximate NN $\hat{\mathbf{q}}$, the distance $\varrho_{\mathbf{q}} = \|\mathbf{q} - \hat{\mathbf{q}}\|$ and multiplicity $\chi_S(\mathbf{q})$ for each query point $\mathbf{q} \in S^e$.

**function** `allNNSearch`(tree $T$):
  **foreach** *leaf* $Q$ *of the tree* $T$ **do**
    **foreach** *point* $\mathbf{q}$ *from leaf* $Q$ **do**
      call `NNSearch`($\mathbf{q}$,$Q$) and store results

**function** `NNSearch`(query point $\mathbf{q}$ from leaf $Q$):
  $\mathbf{p}_{\min} \leftarrow \arg\min_{\mathbf{x} \in Q \setminus \{\mathbf{q}\}} \|\mathbf{q} - \mathbf{x}\|$
  $\varrho_{\min} \leftarrow \|\mathbf{q} - \mathbf{p}_{\min}\|$
  **if** $\varrho_{\min} = 0$ **then**
    **return** $\big(\mathbf{p}_{min}, \varrho_{min}, \chi_S(\mathbf{q})\big)$
  $v \leftarrow V - \|Q\|$
  PQ $\leftarrow$ empty priority queue of node pairs $(Z, Z')$ sorted by $\eta_Z$
  `PushIfBetter(`*Parent*`(Q), Q)`
  **while** $PQ$ not empty and $v > 0$ **do**
    $(Z, Z') \leftarrow$ pop the top element from PQ `// element is removed`
    **if** $\eta_Z \geq \varrho_{min}$ **then**
      **return** $(\mathbf{p}_{min}, \varrho_{min}, 1)$
    **if** $Z$ is a leaf **then**
      $\varrho \leftarrow \min_{\mathbf{x} \in Z} \|\mathbf{q} - \mathbf{x}\|$
      $v \leftarrow v - \|Z\|$
      **if** $\varrho < \varrho_{min}$ **then**
        $\varrho_{\min} \leftarrow \varrho$
        $\mathbf{p}_{\min} \leftarrow \arg\min_{\mathbf{x} \in Z} \|\mathbf{q} - \mathbf{x}\|$
    **else**
      `// Examine yet unseen nodes accessible from` $Z$
      **for** $X$ in $\{Parent, LeftChild, RightChild\}(Z) \setminus \{Z'\}$ **do**
        `PushIfBetter(`$X$`, `$Z$`)`
  **return** $(\mathbf{p}_{min}, \varrho_{min}, 1)$

**function** `PushIfBetter`(new node $X$, previous node $Z$):
  **if** $X = Parent(Z)$ **then**
    $\eta_X = d\big(\mathbf{q}, \mathbb{R}^D \setminus \mathrm{LBB}_Z\big)$
  **else**
    $\eta_X = d\big(\mathbf{q}, \mathrm{TBB}_X\big)$
  **if** $\eta_X < \varrho_{min}$ **then**
    push $(X, Z, \eta_X)$ to PQ

**Algorithm 3**: `UpdateTree`, updating the tree after the points have been moved.

**Input**: Tree $T$ with root $R$ and modified points $S'$
**Output**: Updated tree $T$.

**function** `UpdateTree`(tree root $R$):
  `UpdateNode`($R$)
  `RebuildNode`($R$)

**function** `UpdateNode`(tree node $Q$):
  **if** $Q$ is not a leaf **then**
    `UpdateNode`(*LeftChild(Q)*)
    `UpdateNode`(*RightChild(Q)*)
  move points from $Q$ not inside $\mathrm{LBB}_Q$ to *Parent*($Q$)

**function** `RebuildNode`(tree node $Q$):
  **if** $X$ is a leaf **then**
    compute and store $TBB_X$
  **else**
    **foreach** point **x** in $Q$ **do**
      `//` $m$ `and` $\xi$ `are the splitting dimension and value for` $Q$
      **if** $x_m < \xi$ **then**
        move **x** to *LeftChild*($Q$)
      **else**
        move **x** to *RightChild*($Q$)
    `//` $\|\cdot\|$ `denotes the number of points in a subtree`
    **if** $\max(\|\mathit{LeftChild}(Q)\|, \|\mathit{RightChild}(Q)\|) > (\frac{1}{2} + \delta)\,\|Q\|$ **then**
      $X \leftarrow$ `SplitNode`($Q$)  `// Algorithm 1`
    **else**
      `RebuildNode`$\big($*LeftChild*($Q$)$\big)$
      `RebuildNode`$\big($*RightChild*($Q$)$\big)$
      $\mathrm{TBB}_Q \leftarrow$ combine TBBs of children of $X$

whole tree needs to be rebuilt and the complexity is the complexity of the `BuildTree` operation, $O\big(dn \log(n/L)\big)$.

## 2.6 Implementation notes

Let us examine some of the implementation choices made and possible alternatives. The multisets of points stored at each node are represented as doubly-linked lists. Considering that during the NN search we need to access all points in a leaf and the order is unimportant, it means that all needed operations have a constant complexity per point. As the median search also does not need a random access operation, we conclude that this choice does not increase the total asymptotic complexity of any of the presented algorithms. A previously explored alternative is to implement the multisets as hash tables [61], which according to our experiments seems to be slower.

If a significant number of points with high multiplicity is expected, it might be advantageous to identify them and store the multiplicities explicitly, reducing the time complexity of dealing with a point of multiplicity $M$ from $O(M^2)$ to $O(1)$ and the memory consumption from $O(M)$ to $O(1)$. The detection is cheap, since any point with multiplicity $M > L$ will lead to a degenerate split. This is not implemented in the code evaluated here. Instead, we detect degenerate nodes by noting that their spatial dimension is zero, which needs less overhead and catches the majority of important cases of multiplicity in our data.

We have modified the well-known quicksort-like median finding and splitting algorithm [72, 71], so that if there are several values equal to the median, they are guaranteed to end up in the same (upper) part of the split.

Degenerate splitting happens if more than half of the points in the node is identical with respect to the splitting dimension. In this situation it is possible to consider a splitting immediately above the block of identical values corresponding to the median and test whether this leads to a fairer splitting. However, on our data this made almost no difference and did not offset the additional computation ($O(m)$ operations to find a new splitting value $\xi$).

In the `BuildTree` routine, instead of making each node contain at most $L$ points, we could make it contain *at least* $L$ points. This helps to avoid almost degenerate splits and ensures a bound on the number of visited leaves per query at the expense of sometimes unused median calculation and less efficient handling of multiple points. On our data, this variant seems to be slightly slower.

Memory consumption can be reduced at the expense of some extra calculations. LBB do not have to be stored in the nodes and can be calculated on-the-fly during the top-down tree traversal (for tree building, searching,

and update) using the splitting parameters $m$ and $\xi$ at each node. LBBs can be used instead of TBBs, further reducing the memory consumption and speeding up tree building and update at the expense of deteriorating the search performance.

# 3 Experiments

We have implemented the algorithm in C++ and performed several experiments on an Intel 1.8GHz PCs with 2GB of RAM running Linux to test its practical properties and its performance against some alternative approaches.[1] We are comparing our results (the all-NN search algorithm, denoted BBF) against the brute force algorithm (refered to as 'brute') with time complexity $O(N^2)$ and a state-of-the-art approximate NN search implementation in the ANN library by Arya et al.[30] (referred to as ANN), which uses a balanced box decomposition (BBD) tree. The ANN library was compiled from the available source code and all parameters (except the currently tested one) were left to default. For comparison, we have also implemented the $n$ times repeated NN search using our BBF approach, denoted BBF NNN.

Most experiments are demonstrated with normally and isotropically distributed points because we have found them sufficiently representative. For results with other data distributions, see Section 3.4. Since some methods were not able to process the largest data sets — mostly because it would take an excessive amount of time — these results are omitted in the graphs and tables below. Each value shown is a mean of 10 runs, unless specified otherwise.

## 3.1 Leaf size

The first experiment (Table 1) studies the effect of the parameter $L$ as a function of the dimensionality $d$. The parameter $L$ determines the maximum number of points in a leaf (Section 2.2). We use $n = 10^5$ and we show the timings for building the tree, one all-NN search (corresponding to $n = 10^5$ single NN queries), and the sum of the two times. We can see that the build times decrease with increasing $L$ for all $d$, because the number of nodes decreases. On the other hand, the search times initially decrease with increasing $L$ up to an optimum balance between the number of nodes to be

[1] For organizational reasons, we unfortunately were not able to use one single machine for all the experiments, instead we used a set of similar ones. The times between different experiments might therefore not be directly commensurable.

searched and the number of nodes in each point, then start to slowly increase again. The total time is a superposition of these two effects, for one complete all-NN search per tree build we can see that in low dimensions ($d \lessapprox 4$) it is advantageous to use larger values of $L$ ($L \approx 30$), while as the dimensionality $d$ grows, the optimal $L$ decreases to $L = 4$ for $d = 20$. The optimum is not sharp, so the precise choice is not critical. For approximate search, we found it beneficial to choose slightly higher values of $L$, such as $L = 30$. The values of $L$ determined here were used in the rest of the experiments.

## 3.2 Exact search

The second set of experiments demonstrates the dependence of the total time to find all NNs on the number of points $n$ for different dimensions $d$ and the four tested methods (brute force, ANN, BBF NNN, and BBF). We can see in Figure 3 that while the time complexity of the brute force method is quadratic as expected, the time complexity of the other methods is subquadratic — a linear regression on the logarithmic plots shown estimates the complexity between $O(n^{1.1})$ for $d = 1$ and the BBF method and $O(n^{1.8})$ for $d = 20$ and the ANN method. All three subquadratic methods outperform the brute force search and the BBF method is almost always the best of them but the difference narrows for higher $d$ (compare Figure 3ef).

A more detailed comparison between the three subquadratic methods is shown in Table 2. We can see that for $n \geq 10^5$, our BBF method outperforms the ANN for all $d$. The difference is very pronounced in low dimensions (a factor of 7 for $d = 1$) and decreases with increasing $d$. This seems to be a general pattern: in higher dimensions all methods struggle and their differences get smaller.

The comparison between (all-NN) BBF and NNN BBF is interesting because it indicates the gain due to an all-NN search with respect to repeated NN search. Since for higher $d$ ANN in many cases outperforms NNN BBF, we can hypothesize that combining the all-NN search with the BBD tree used in ANN could lead to an even faster algorithm. On the other hand, ANN is the slowest method for small $d$, probably because of a larger overhead.

## 3.3 Approximative search

The third set of experiments compares the performance of the ANN and BBF methods for approximative search by examining their time-accuracy trade-off. The accuracy of the NN search is evaluated by using the reported approximate NNs to estimate the entropy of the given set of points using

Table 1: Each box shows from the top to bottom the times in seconds to build the tree from $n = 10^5$ normally and isotropically distributed points, to perform one all-NN search, and the total time (the sum of these two times), respectively, as a function of the space dimension $d$ and the number of points per node $L$. The shortest total times for each dimension $d$ are typeset in bold. Each number is a mean of ten experiments.

| $L/d$ | 1 | 2 | 3 | 4 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|---|---|
| | 0.209 | 0.229 | 0.242 | 0.253 | 0.264 | 0.352 | 0.405 | 0.472 |
| 1 | 0.108 | 0.216 | 0.439 | 0.855 | 1.557 | 18.502 | 104.038 | 318.329 |
| | 0.318 | 0.445 | 0.681 | 1.108 | 1.821 | 18.854 | 104.443 | 318.801 |
| | 0.150 | 0.170 | 0.181 | 0.190 | 0.200 | 0.275 | 0.334 | 0.391 |
| 2 | 0.058 | 0.142 | 0.315 | 0.639 | 1.191 | 15.190 | 86.276 | 263.688 |
| | 0.208 | 0.312 | 0.495 | 0.829 | 1.391 | 15.465 | 86.610 | 264.079 |
| | 0.090 | 0.108 | 0.118 | 0.127 | 0.136 | 0.191 | 0.243 | 0.288 |
| 4 | 0.031 | 0.093 | 0.233 | 0.508 | 0.990 | 13.725 | 80.388 | 248.478 |
| | 0.122 | 0.201 | 0.352 | 0.635 | 1.126 | 13.916 | **80.631** | **248.766** |
| | 0.062 | 0.078 | 0.087 | 0.095 | 0.103 | 0.155 | 0.202 | 0.239 |
| 6 | 0.025 | 0.076 | 0.203 | 0.455 | 0.906 | 13.506 | 82.113 | 255.595 |
| | 0.087 | 0.154 | 0.290 | 0.550 | 1.009 | 13.661 | 82.315 | 255.834 |
| | 0.058 | 0.075 | 0.084 | 0.092 | 0.099 | 0.153 | 0.197 | 0.232 |
| 8 | 0.024 | 0.073 | 0.199 | 0.449 | 0.903 | 13.483 | 82.643 | 256.880 |
| | 0.083 | 0.148 | 0.282 | 0.540 | 1.002 | **13.636** | 82.840 | 257.112 |
| | 0.058 | 0.074 | 0.083 | 0.092 | 0.100 | 0.153 | 0.197 | 0.232 |
| 10 | 0.024 | 0.073 | 0.198 | 0.451 | 0.900 | 13.527 | 82.720 | 256.477 |
| | 0.083 | 0.148 | 0.282 | 0.543 | 1.000 | 13.680 | 82.917 | 256.709 |
| | 0.040 | 0.054 | 0.062 | 0.069 | 0.076 | 0.123 | 0.163 | 0.198 |
| 15 | 0.023 | 0.065 | 0.177 | 0.422 | 0.865 | 13.900 | 89.266 | 269.407 |
| | 0.063 | 0.119 | 0.239 | 0.491 | **0.941** | 14.024 | 89.429 | 269.605 |
| | 0.040 | 0.055 | 0.062 | 0.069 | 0.076 | 0.125 | 0.163 | 0.197 |
| 20 | 0.024 | 0.065 | 0.176 | 0.423 | 0.867 | 13.888 | 89.105 | 269.247 |
| | 0.063 | 0.119 | 0.238 | 0.492 | 0.943 | 14.013 | 89.268 | 269.444 |
| | 0.031 | 0.044 | 0.050 | 0.056 | 0.063 | 0.107 | 0.142 | 0.171 |
| 25 | 0.027 | 0.066 | 0.180 | 0.422 | 0.881 | 14.660 | 96.481 | 273.027 |
| | 0.058 | 0.110 | **0.230** | **0.478** | 0.944 | 14.767 | 96.623 | 273.198 |
| | 0.031 | 0.044 | 0.051 | 0.057 | 0.063 | 0.106 | 0.143 | 0.171 |
| 30 | 0.027 | 0.066 | 0.180 | 0.424 | 0.879 | 14.649 | 96.182 | 273.880 |
| | **0.058** | **0.110** | 0.230 | 0.480 | 0.942 | 14.756 | 96.325 | 274.051 |

Table 2: The total time in seconds to find all NNs as a function of the number of points $n$ and dimension $d$ for the three subquadratic methods. The shortest time for each $n$ and $d$ is typeset in bold.

| $n$ | $d$ | 1 | 2 | 3 | 4 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|---|---|---|
| $10^4$ | BBF | **0.01** | **0.01** | **0.04** | **0.07** | 0.12 | 0.84 | 2.52 | 7.74 |
| | NNN BBF | 0.02 | 0.03 | 0.06 | 0.10 | 0.16 | 0.97 | 2.67 | 8.01 |
| | ANN | 0.04 | 0.04 | 0.06 | 0.08 | **0.12** | **0.80** | **2.51** | **7.43** |
| $10^5$ | BBF | **0.07** | **0.13** | **0.50** | **0.91** | **1.58** | **20.32** | **92.46** | **370.54** |
| | NNN BBF | 0.34 | 0.48 | 0.91 | 1.70 | 2.75 | 26.32 | 97.56 | 372.00 |
| | ANN | 0.50 | 0.60 | 0.93 | 1.48 | 2.26 | 25.04 | 107.24 | 434.44 |
| $10^6$ | BBF | **1.10** | **1.91** | **5.33** | **10.22** | **17.56** | **335.54** | **2562.78** | — |
| | NNN BBF | 6.10 | 7.77 | 11.75 | 22.44 | 34.81 | 417.34 | 2654.09 | — |
| | ANN | 8.16 | 9.05 | 13.04 | 20.66 | 30.64 | 364.80 | 2989.96 | — |

the KL NN entropy estimator (Section 1.3), which is our intended application. We have used $n = 10^5$ normally and isotropically distributed points in dimensions $d = 3 \sim 20$ which have allowed the entropy to be calculated analytically. The times reported include building the tree and finding $n$ approximate NN neighbors. A relative mean square error of the estimated entropy with respect to the true entropy is shown. For the BBF method we have varied the parameter $V$ which determines the maximum number of examined points (Section 2.4.1) in the range $V = 3 \sim 10^5$. For the ANN method we have varied the parameter $\epsilon$ which determines the maximum allowed relative error [30] (Section 1.1) in the range $\epsilon = 10^{-2} \sim 10^2$. The values for both $V$ and $\epsilon$ are chosen to safely span the complete range between a very approximative search, when only a couple of points are examined, and a very accurate search, when no approximation is made.

From the results shown in Figure 4 we can see that our BBF method outperforms ANN for small $d$. Starting from about $d = 7$, BBF is better for operating points privileging shorter times and larger errors, while ANN performs better at longer times and smaller errors.

## 3.4 Effect of different data distributions

The experiments so far were performed with normally distributed points. We will now show the performance of the methods for other distributions, mostly taken from [30]. Here is a list of the distributions used:

**Normal** — isotropic normal distribution, used in all preceding experiments.

**Uniform** — a uniform isotropic distribution.

**Correlated normal** (`corrnormal`) — The data points $\mathbf{x}_i$ are calculated according to the formula $\mathbf{x}_i = 0.9\mathbf{x}_{i-1} + 0.1\mathbf{w}_i$, with zero mean i.i.d. normally distributed $\mathbf{w}_i$.

**Correlated uniform** (`corruniform`) — As for `corrnormal` but for uniformly distributed $\mathbf{w}_i$.

**Clustered** — Ten cluster centers are chosen uniformly from $[0, 1]^d$. Each of the $n$ points is a random additive perturbation of a randomly chosen cluster center. The perturbation is normal, with standard deviation $\sigma = 0.05$ in each dimension.

**Subspace** — The points lie on a randomly chosen linear subspace of dimension one with random additive normal perturbation with $\sigma = 0.05$.

**Image neighborhood** (`imgneighb`) — This distribution is inspired by our target application, namely by calculating a mutual information image similarity criterion [29] (Section 1.4). It is based on the 8-bit green channel of the $512 \times 512$ Lena image. For each data point to be generated, we randomly select a pixel location in this image. The $d$ values needed are the intensities on a discrete spiral centered at the chosen pixel. We have also added a small uniformly distributed perturbation (with amplitude 0.01) for the benefit of ANN, which fails otherwise (with a segmentation fault) probably because the data contains multiple points.

The measured times for the exact all-NN search using BBF and ANN methods are reported in Table 3 for $n = 10^6$ and several values of $d$. We see that BBF in almost all cases performs better than ANN, especially for the `imgneighb` distribution which was our primary aim.

## 3.5 Incremental update

The last experiment (Table 4) evaluates the effectiveness of the tree update operation (Section 2.5). We have generated $n = 10^6$ uniformly distributed points from the hyperinterval $[-1; 1]^d$ with $d = 5$ and then perturbed them by adding another set of uniformly distributed points from $[-\sigma, \sigma]^d$. We compare two methods of obtaining all exact NNs for the perturbed dataset. First, we build the tree directly from the perturbed dataset using `BuildTree` (Algorithm 1) and run the search, this corresponds to the time $T_{b+s} = T_b + T_s$ in Table 4. Second, we build the tree for the unperturbed dataset, update it using the `UpdateTree` procedure (Algorithm 3) for the perturbed dataset,

Table 3: Comparison of the tree build, search, and total times for an all-NN search for the BBF and ANN methods depending on the point distribution, for $n = 10^6$. The shortest total time for each configuration (dimension $d$ and a distribution type) is typeset in bold.

| | | BBF | | | ANN | | |
|---|---|---|---|---|---|---|---|
| type | $d$ | bld | search | tot | bld | search | tot |
| normal | 2 | 1.04 | 0.68 | **1.72** | 1.46 | 8.83 | 10.29 |
| normal | 5 | 1.48 | 10.21 | **11.69** | 2.99 | 24.55 | 27.54 |
| normal | 10 | 2.53 | 249.99 | **252.52** | 6.08 | 288.74 | 294.82 |
| normal | 20 | 3.57 | 7956.07 | **7959.64** | 7.88 | 9622.11 | 9629.99 |
| uniform | 2 | 1.03 | 0.69 | **1.72** | 1.31 | 7.51 | 8.82 |
| uniform | 5 | 1.47 | 7.94 | **9.41** | 2.48 | 20.64 | 23.12 |
| uniform | 10 | 2.53 | 110.73 | **113.26** | 5.40 | 141.39 | 146.79 |
| uniform | 20 | 3.52 | 2755.73 | **2759.25** | 10.26 | 2911.56 | 2921.82 |
| corrnormal | 2 | 0.92 | 0.67 | **1.59** | 1.26 | 7.53 | 8.79 |
| corrnormal | 5 | 1.33 | 9.90 | **11.23** | 2.57 | 21.25 | 23.82 |
| corrnormal | 10 | 2.22 | 234.99 | 237.21 | 5.42 | 193.28 | **198.70** |
| corrnormal | 20 | 3.26 | 3436.89 | **3440.15** | 7.29 | 3584.14 | 3591.43 |
| corruniform | 2 | 0.95 | 0.67 | **1.62** | 1.23 | 8.41 | 9.64 |
| corruniform | 5 | 1.34 | 9.75 | **11.09** | 2.58 | 21.82 | 24.40 |
| corruniform | 10 | 2.24 | 222.89 | 225.13 | 4.82 | 195.38 | **200.20** |
| corruniform | 20 | 3.29 | 1834.14 | **1837.43** | 7.13 | 1859.78 | 1866.91 |
| clustered | 2 | 1.03 | 0.68 | **1.71** | 1.47 | 8.65 | 10.12 |
| clustered | 5 | 1.47 | 9.18 | **10.65** | 3.67 | 22.58 | 26.25 |
| clustered | 10 | 2.64 | 138.30 | **140.94** | 7.18 | 200.89 | 208.07 |
| clustered | 20 | 3.49 | 2527.54 | **2531.03** | 13.47 | 2818.86 | 2832.33 |
| subspace | 2 | 1.05 | 0.68 | **1.73** | 1.43 | 7.54 | 8.97 |
| subspace | 5 | 1.44 | 9.69 | **11.13** | 3.03 | 22.18 | 25.21 |
| subspace | 10 | 2.52 | 186.68 | **189.20** | 6.59 | 238.72 | 245.31 |
| subspace | 20 | 3.55 | 4656.80 | **4660.35** | 11.04 | 5349.66 | 5360.70 |
| imgneighb | 2 | 1.00 | 0.69 | **1.69** | 1.67 | 7.34 | 9.01 |
| imgneighb | 5 | 1.52 | 2.57 | **4.09** | 4.97 | 9.39 | 14.36 |
| imgneighb | 10 | 2.54 | 5.36 | **7.90** | 10.80 | 12.00 | 22.80 |
| imgneighb | 20 | 3.54 | 14.47 | **18.01** | 27.44 | 21.75 | 49.19 |

and then run the search; we report the update and search times as $T_{u+s'} = T_u + T_{s'}$. We can see that it is indeed advantageous to update the tree instead of rebuilding it anew, as long we allow some unbalance $\delta$. The time savings in terms of the total time ($T_{b+s}$, $T_{u+s'}$) are relatively modest but the savings in terms of the update versus build times ($T_u$, $T_s$) are already more important, around 50%. This is relevant as in practice mostly approximative search will be used which reduces the search times by one or several orders of magnitude, making the build or update times dominate.

The optimal value of $\delta$ is a trade-off between keeping the tree balanced and avoiding rebuilding it frequently. We observe that when the changes of point positions are random (as in this particular case), the tree stays relatively well balanced and changing $\delta$ has very little effect. Somewhat unexpectedly, the post-update $T_{s'}$ search times are consistently longer than the search times $T_s$ on a freshly built tree, especially for $\delta = 0$. We believe this is because of worse memory fragmentation and locality of reference after running the `UpdateTree` procedure; we have verified that for $\delta = 0$, the tree produced by `UpdateTree` is structurally identical to a tree produced by `BuildTree` directly.

# 4 Conclusions

We have described a new algorithm to find all exact and approximate nearest neighbors in a set of points. The algorithm is based on a $k$-d tree, with several modifications: The tree nodes are augmented by tight bounding boxes (TBBs) to improve pruning. The search proceeds in a best-bin first (BBF) manner and can be stopped prematurely if an approximate NN is acceptable, with the corresponding speed gain. The tree can be efficiently updated when the point coordinates change. The algorithm can work with datasets containing multiple points, the multiplicities are correctly handled and reported. A dedicated all-NN search strategy is used instead of repeating $n$ times the standard NN search, with corresponding computational savings.

The algorithm development was motivated by a particular application — evaluating entropy using a Kozachenko-Leonenko (KL) NN entropy estimator [18] for a mutual information based image similarity measure for image registration [61]. The KL estimator has some very beneficial properties and we hope that a sufficiently fast all-NN search algorithm such as ours could trigger its more widespread use. We have shown experimentally that our algorithm compares favorably with the state of the art approximate nearest neighbor search library ANN [30] and we expect it to find a large range of applications.

Table 4: After generating $n = 10^6$ uniformly distributed points in dimension $d = 5$ we perturb them by adding a uniformly distributed perturbation from interval $[-\sigma, \sigma]^d$. We show the build ($T_b$) and all-NN search ($T_s$) times for the perturbed dataset as well as the time ($T_u$) needed to update a tree created for the first dataset and the search time ($T_{s'}$) using the updated tree. All times are in seconds. We also show the sums $T_{b+s} = T_b + T_s$ and $T_{u+s'} = T_u + T_{s'}$. The best of the two times for the pairs $T_{b+s}, T_{u+s'}$ and $T_b, T_u$ is typeset in bold. The parameter $\delta$ controls the allowed unbalance of the tree. The standard deviation of the times is in the range $0.01 \sim 0.05$ s.

| $\sigma$ | $\delta$ | $T_b$ | $T_s$ | $T_u$ | $T_{s'}$ | $T_{b+s}$ | $T_{u+s'}$ |
|---|---|---|---|---|---|---|---|
| 0.001 | 0.0 | **1.52** | 7.94 | 1.69 | 8.56 | **9.46** | 10.25 |
| 0.001 | 0.1 | 1.54 | 7.96 | **0.31** | 8.11 | 9.49 | **8.42** |
| 0.001 | 0.2 | 1.51 | 7.96 | **0.32** | 8.09 | 9.46 | **8.40** |
| 0.001 | 0.3 | 1.53 | 7.96 | **0.32** | 8.08 | 9.50 | **8.40** |
| 0.001 | 0.4 | 1.53 | 7.96 | **0.31** | 8.08 | 9.49 | **8.39** |
| 0.010 | 0.0 | **1.51** | 7.99 | 1.83 | 8.71 | **9.50** | 10.54 |
| 0.010 | 0.1 | 1.52 | 8.03 | **0.39** | 8.12 | 9.54 | **8.52** |
| 0.010 | 0.2 | 1.51 | 8.02 | **0.40** | 8.09 | 9.53 | **8.49** |
| 0.010 | 0.3 | 1.53 | 8.00 | **0.39** | 8.13 | 9.53 | **8.52** |
| 0.010 | 0.4 | 1.50 | 8.02 | **0.40** | 8.13 | 9.52 | **8.52** |
| 0.100 | 0.0 | **1.52** | 8.74 | 2.30 | 9.78 | **10.26** | 12.07 |
| 0.100 | 0.1 | 1.53 | 8.72 | **0.87** | 9.16 | 10.24 | **10.03** |
| 0.100 | 0.2 | 1.51 | 8.74 | **0.83** | 9.21 | 10.26 | **10.04** |
| 0.100 | 0.3 | 1.53 | 8.71 | **0.83** | 9.30 | 10.24 | **10.13** |
| 0.100 | 0.4 | 1.53 | 8.71 | **0.82** | 9.25 | 10.24 | **10.07** |

General nearest neighbor search in high dimensions is a hard problem. But it is by no means hopeless, especially if approximate search is acceptable.

# References


[1] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, From data mining to knowledge discovery in databases, AI Magazine 17 (1996) 37–54.
URL `citeseer.ist.psu.edu/fayyad96from.html`

[2] S. Wess, K.-D. Althoff, G. Derwand, Using $k$-D trees to improve the retrieval step in case-based reasoning, in: EWCBR, 1993, pp. 167–181.
URL `citeseer.ist.psu.edu/wess93using.html`

[3] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, Z. Protopapas, Fast nearest neighbor search in medical image databases, in: The VLDB Journal, 1996, pp. 215–226.
URL `citeseer.ist.psu.edu/article/korn96fast.html`

[4] M. S. Waterman, Introduction to Computational Biology, Chapman and Hall, 2000.
URL `http://www-hto.usc.edu/books/msw/msg/index.html`

[5] S. W. Park, L. Linsen, O. Kreylos, J. D. Owens, B. Hamann, Discrete sibson interpolation, IEEE Transactions on Visualization and Computer Graphics 12 (2) (2006) 243–253. doi:http://dx.doi.org/10.1109/TVCG.2006.27.

[6] R. O. Duda, P. E. Hart, D. G. Stork, Pattern classification, 2nd Edition, Wiley Interscience Publication, John Wiley, New York, 2001.

[7] T. M. Cover, P. E. Hart, Nearest neighbor pattern classification, IEEE Transactions on Information Theory 13 (1967) 21–27.

[8] A. Gersho, R. M. Gray, Vector quantization and signal compression, Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[9] R. M. Gray, D. L. Neuhoff, Quantization, IEEE Trans. Inform. Theory 44 (1993) 2325–2383.

[10] D. E. Knuth, The Art of Computer Programming, Vol 3: Sorting and Searching, 2nd Edition, Addison-Wesley, Reading, Mass., 1998.

[11] P. M. Vaidya, An $O(n \log n)$ algorithm for the all-nearest-neighbors problem, Discrete Comput. Geom. 4 (2) (1989) 101–115.

[12] K. L. Clarkson, Fast algorithms for the all nearest neighbors problem, in: Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci., 1983, pp. 226–232.

[13] L. J. Buturovic, Improving $k$-nearest neighbor density and error estimates, Pattern Recognition 26 (4) (1993) 611–616. doi:http://dx.doi.org/10.1016/0031-3203(93)90114-C.

[14] J. Costa, A. Hero, Manifold learning using euclidean $k$-nearest neighbor graphs, in: Proc. IEEE Int'l Conf. Acoustic Speech and Signal Processing, Vol. 3, 2004, pp. 988–991.

[15] K. M. Carter, R. Raich, A. O. Hero, On local dimension estimation and its applications, IEEE Trans. Signal ProcessingIn print.

[16] A. O. Hero, B. Ma, O. Michel, J. Gorman, Applications of entropic spanning graphs, IEEE Signal Proc. Magazine 19 (5) (2002) 85–95. URL `http://www.eecs.umich.edu/~hero/`

[17] J. Beirlant, E. J. Dudewicz, G. L., E. C. van der Meulen, Nonparametric entropy estimation: an overview, International J. Math. Stat. Sci. (6) (1997) 17–39.

[18] L. F. Kozachenko, N. N. Leonenko, On statistical estimation of entropy of random vector, Probl. Inf. Trans. 23 (9), (in Russian).

[19] J. D. Victor, Binless strategies for estimation of information from neural data, Physical Review E 66 (5) (2002) 051903(15).

[20] Q. Wang, S. Kulkarni, S. Verdu, A nearest-neighbor approach to estimating divergence between continuous random vectors, in: IEEE Int. Symp. Information Theory, 2006, pp. 242–246.

[21] A. Kraskov, H. Stögbauer, P. Grassberger, Estimating mutual information, Physical Review E (69 066138).

[22] P. Viola, W. M. I. Wells, Alignment by maximization of mutual information, International Journal of Computer Vision (2) (1997) 137–154.

[23] J. Pluim, J. B. A. Maintz, M. A. Viergever, Mutual-information-based registration of medical images: A survey, IEEE Transactions on Medical Imaging 22 (8) (2003) 986–1004.

[24] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, P. Suetens, Multimodality image registration by maximization of mutual information, IEEE Transactions on Medical Imaging 16 (2) (1997) 187–198.

[25] P. Thévenaz, M. Unser, Optimization of mutual information for multiresolution image registration, IEEE Transactions on Image Processing 9 (12) (2000) 2083–2099.

[26] D. Rueckert, M. J. Clarkson, D. L. G. Hill, D. J. Hawkes, Non-rigid registration using higher-order mutual information, in: Proceedings of SPIE Medical Imaging 2000: Image Processing, 2000, pp. 438–447.

[27] J. P. W. Pluim, J. B. A. Maintz, M. A. Viergever, Image registration by maximization of combined mutual information and gradient information, IEEE Transactions Med. Imag. 19 (8).

[28] D. B. Russakoff, C. Tomasi, T. Rohlfing, C. R. J. Maurer, Image similarity using mutual information of regions, in: T. Pajdla, J. Matas (Eds.), Proceedings of the 8th European Conference on Computer Vision (ECCV), no. 3023 in LNCS, Springer, 2004, pp. 596–607.

[29] J. Kybic, High-dimensional mutual information estimation for image registration, in: ICIP'04: Proceedings of the 2004 IEEE International Conference on Image Processing, IEEE Computer Society, 445 Hoes Lane, Piscataway, NJ, U.S.A., 2004.
URL `ftp://cmp.felk.cvut.cz/pub/cmp/articles/kybic/Kybic-ICIP2004.pdf`

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

[31] W. D. Blizzard, Multiset theory, Notre Dame J. Formal Logic (1) (1988) 36–66.

[32] D. E. Knuth, The Art of Computer programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, 1998.

[33] P. Tsaparas, Nearest neighbor search in multidimensional spaces, Tech. Rep. 31902, Dept. of Computer Science, University of Toronto, Canada, qualifying Depth Oral Report (1999).
URL `citeseer.ist.psu.edu/tsaparas99nearest.html`

[34] J. H. Friedman, J. L. Baskett, L. J. Shustek, An algorithm for finding nearest neighbors, IEEE Trans. Comput. 24 (1975) 1000–1006.

[35] T. P. Yunck, A technique to identify nearest neighbors, IEEE Trans. Systems, Man, and Cybernetics 6 (10) (1976) 678–683.

[36] S. A. Nene, S. K. Nayar, A simple algorithm for nearest neighbor search in high dimensions, IEEE Trans. Pattern Anal. Mach. Intell. 19 (9) (1997) 989–1003. doi:http://dx.doi.org/10.1109/34.615448.

[37] M. Smid, Closest-Point Problems in Computational Geometry, North Holand, 2000.

[38] H. Samet, The quadtree and related hierarchical data structures, ACM Comput. Surv. 16 (2) (1984) 187–260. doi:http://doi.acm.org/10.1145/356924.356930.

[39] D. Eppstein, M. T. Goodrich, J. Z. Sun, The skip quadtree: a simple dynamic data structure for multidimensional data, in: SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry, ACM Press, New York, NY, USA, 2005, pp. 296–305. doi:http://doi.acm.org/10.1145/1064092.1064138.

[40] D. A. White, R. Jain, Similarity indexing with the SS-tree, in: ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1996, pp. 516–523.

[41] N. Katayama, S. Satoh, SR-tree: An index structure for nearest neighbor searching of high-dimensional point data, Transactions of the Institute of Electronics, Information and Communication Engineers J80-D-I (8) (1997) 703–717.

[42] J. L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517. doi:http://doi.acm.org/10.1145/361002.361007.

[43] J. H. Freidman, J. L. Bentley, R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, ACM Trans. Math. Softw. 3 (3) (1977) 209–226. doi:http://doi.acm.org/10.1145/355744.355745.

[44] J. S. Beis, D. G. Lowe, Shape indexing using approximate nearest-neighbour search in high-dimensional spaces, in: Proceedings of Conference on Computer Vision and Pattern Recognition, 1997, pp. 1000–1006.

[45] P. K. Agarwal, J. Gao, L. J. Guibas, Kinetic medians and $k$-d-trees, in: Proc. 10th European Sympos. Algorithms, 2002, pp. 5–16.

[46] N. Sample, M. Haines, M. Arnold, T. Purcell, Optimizing search strategies in k-d trees, in: 5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers (CSCC 2001), 2001.
URL `http://ilpubs.stanford.edu:8090/723/`

[47] P. B. Callahan, S. R. Kosaraju, A decomposition of multi-dimensional point-sets with applications to $k$-nearest-neighbors and $n$-body potential fields, in: Proceedings 24th Annual AMC Symposium on the Theory of Computing, 1992, pp. 546–556.
URL `citeseer.nj.nec.com/callahan92decomposition.html`

[48] P. Indyk, Nearest neighbors in high-dimensional spaces, in: J. E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, chapter 39, CRC Press, 2004, 2rd edition.
URL `citeseer.ist.psu.edu/indyk04nearest.html`

[49] T. Liu, A. W. Moore, A. Gray, K. Yang, An investigation of practical approximate nearest neighbor algorithms, in: L. K. Saul, Y. Weiss, L. Bottou (Eds.), Advances in Neural Information Processing Systems 17, MIT Press, Cambridge, MA, 2005, pp. 825–832.

[50] E. Kushilevitz, R. Ostrovsky, Y. Rabani, Efficient search for approximate nearest neighbor in high dimensional spaces, SIAM J. Comput. 30 (2) (2000) 457–474.

[51] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in: Proc. of 30th STOC, 1998, pp. 604–613.
URL `citeseer.ist.psu.edu/article/indyk98approximate.html`

[52] R. Panigrahy, Entropy based nearest neighbor search in high dimensions, in: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, ACM Press, New York, NY, USA, 2006, pp. 1186–1195. doi:http://doi.acm.org/10.1145/1109557.1109688.

[53] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, Communications of the ACM 51 (1) (2008) 117–122.

[54] M. Datar, N. Immorlica, P. Indyk, V. S. Mirrokni, Locality-sensitive hashing scheme based on $p$-stable distributions, in: SCG '04: Proceedings of the twentieth annual symposium on Computational geometry, ACM Press, New York, NY, USA, 2004, pp. 253–262. doi:http://doi.acm.org/10.1145/997817.997857.

[55] P. K. Agarwal, D. Eppstein, J. Matoušek, Dynamic half-space reporting, geometric optimization, and minimum spanning trees, in: IEEE Symposium on Foundations of Computer Science, 1992, pp. 80–89.
URL `citeseer.ist.psu.edu/agarwal92dynamic.html`

[56] Y. Chiang, R. Tamassia, Dynamic algorithms in computational geometry, Proc. of the IEEE 80 (9) (1992) 362–381, special Issue on Computational Geometry.

[57] P. Bozanis, A. Nanopoulos, Y. Manopoulos, LR-tree: a logarithmic decomposable spatial index method, The computer journal 46 (3) (2003) 319–331.

[58] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, A. Y. Wu, Computing nearest neighbors for moving points and applications to clustering, in: Symposium on Discrete Algorithms, 1999, pp. 931–932.
URL `citeseer.ist.psu.edu/kanungo99computing.html`

[59] J. K. Uhlmann, Satisfying general proximity/similarity queries with metric trees, Information Processing Letters 40 (1991) 175–179.

[60] P. N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1993.

[61] J. Kybic, Incremental updating of nearest neighbor-based high-dimensional entropy estimation, in: P. Duhamel, L. Vandendorpe (Eds.), ICASSP2006, IEEE, Toulouse, France, 2006, pp. III–804, DVD proceedings.
URL `ftp://cmp.felk.cvut.cz/pub/cmp/articles/kybic/Kybic-ICASSP2006.pdf`

[62] J. Kybic, High-dimensional entropy estimation for finite accuracy data: $R$-NN entropy estimator, in: N. Karssemeijer, B. Lelieveldt (Eds.), IPMI2007: Information Processing in Medical Imaging, 20th International Conference, Springer, Berlin, Heidelberg, Germany, 2007,

pp. 569–580.
URL ftp://cmp.felk.cvut.cz/pub/cmp/articles/kybic/Kybic-IPMI2007-color.pdf

[63] V. Mergel, On some properties of Kozachenko-Leonenko estimates and maximum entropy principle in goodness of fit test constructions, in: AIP Conf. Proc. bayesian inference and maximum entropy methods in science and engineering, Vol. 617, 2002, pp. 174–191. doi:10.1063/1.1477047.

[64] D. J. C. MacKay, Information Theory, Inference, and Learning Algorithms, Cambridge University Press, 2003, available from http://www.inference.phy.cam.ac.uk/mackay/itila/.
URL http://www.cambridge.org/0521642981

[65] B. Zitová, J. Flusser, Image registration methods: a survey, Image and Vision Computing (21) (2003) 977–1000.

[66] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, P. Suetens, Multimodality image registration maximization of mutual information, in: MMBIA '96: Proceedings of the 1996 Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA '96), IEEE Computer Society, Washington, DC, USA, 1996, p. 14.

[67] D. B. Russakoff, C. Tomasi, T. Rohlfing, C. R. M. Jr., Image similarity using mutual information of regions, in: ECCV (3), 2004, pp. 596–607.

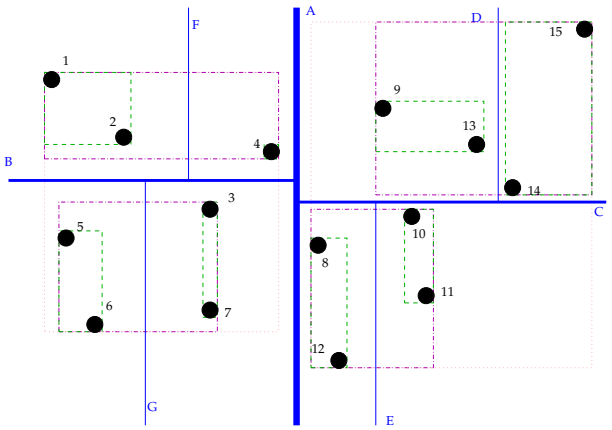
[68] P. Viola, W. M. W. III, Alignment by maximization of mutual information, in: ICCV '95: Proceedings of the Fifth International Conference on Computer Vision, IEEE Computer Society, Washington, DC, USA, 1995, p. 16.

[69] M. R. Sabuncu, P. J. Ramadge, Spatial information in entropy-based image registration., in: J. C. Gee, J. B. A. Maintz, M. W. Vannier (Eds.), WBIR, Vol. 2717 of Lecture Notes in Computer Science, Springer, 2003, pp. 132–141.

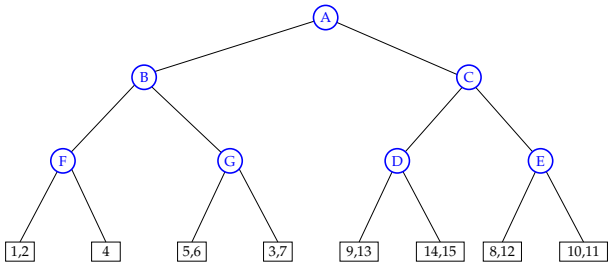
[70] F. P. Preparata, M. I. Shamos, Computational geometry: An introduction, Texts and Monographs in Computer Science, Springer-Verlag, 1985.

[71] R. Sedgewick, Algorithms, Addison-Wesley, Reading, MA, 1988.

[72] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, New York, NY, USA, 1992.

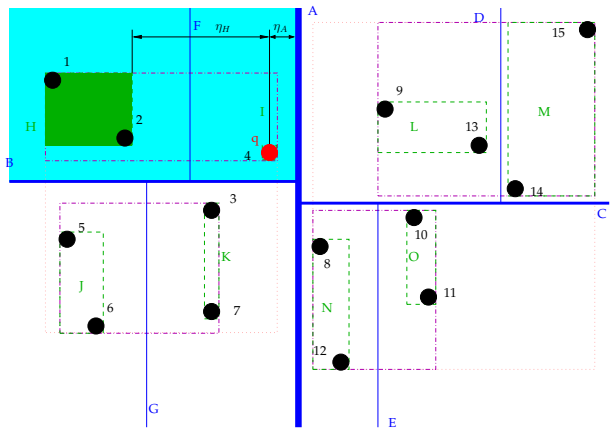



(a)

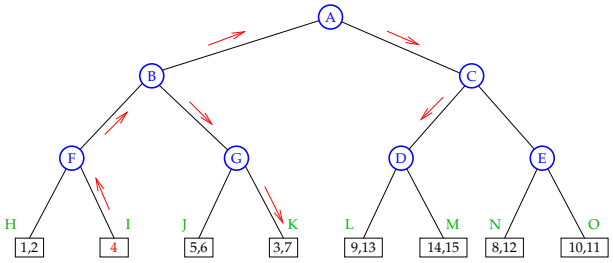



(b)

Figure 1: A $k$-d tree example in a 2D space. (a) Solid blue lines marked by uppercase letters represent the splitting hyperplanes hierarchically subdividing the space into loose bounding boxes (LBB) corresponding to tree nodes. Dashed lines show the tight bounding boxes (TBB). (b) The $k$-d tree itself, with round non-leaf nodes marked by the corresponding dividing hyperplanes and rectangular leaf nodes each containing a set of data points denoted by numbers.




(a)




(b)

Figure 2: The same $k$-d tree as in Figure 1 with an example of a NN search. In the top image (a) we have marked the query point $\mathbf{q}$ (number 4) by a red dot. We have also marked the distance $\eta_H$ from $\mathbf{q}$ to $\mathrm{TBB}_H$ (in green). The distance from $\mathbf{q}$ to the $\mathrm{LBB}_C$ (in cyan) is denoted $\eta_A$, as it is a distance to a yet unexplored part of the tree accessible through $A$ (see the text for more details). On the bottom graph (b) we can observe how a query for the NN of point $\mathbf{q}$ (point number 4) proceeds, starting from the leaf $I$, visiting nodes $F$, $B$, $A$, $C$, $D$, $G$, and $K$, in that order (marked by red arrows), always going to the node with the currently smallest distance bound $\eta$, until the nearest neighbor, point number 3 in leaf $K$ is finally found. All other paths are pruned.

Figure 3: The total time in seconds to find all NNs as a function of the number of points $n$ for different dimensions $d$ for (a) the brute force search, (b) ANN library, (c) our BBF method with $n$ times repeating a simple NN search, and (d) our BBF method with an all-NN search. The times include building the trees. Graphs (e) and (f) show the comparison of different methods for $d = 5$ and $d = 15$, respectively.

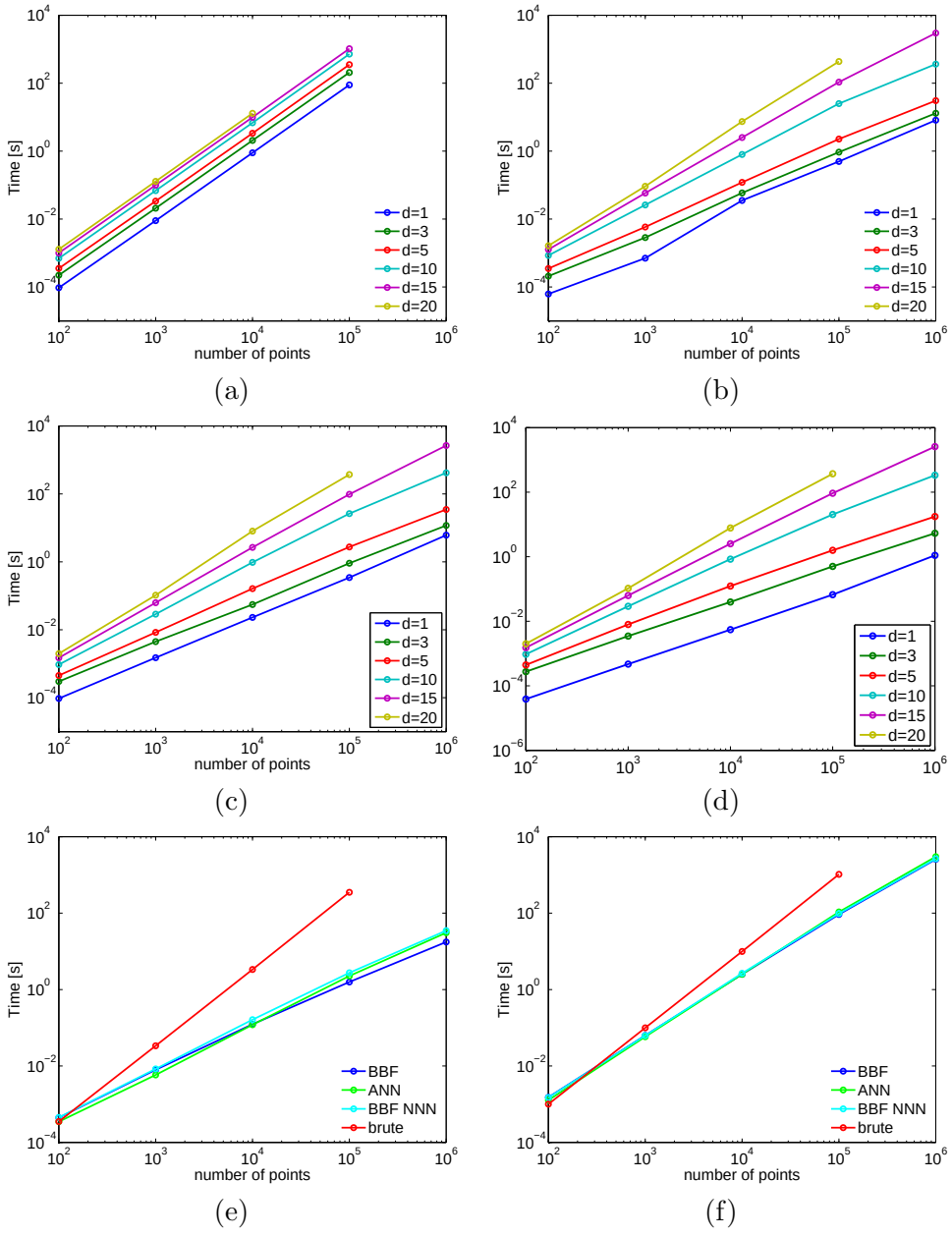


Figure 4: Relative mean square error of entropy estimation versus elapsed time for BBF and ANN methods and the KL NN entropy estimator. There were $n = 10^5$ normally distributed points in dimensions $d = 3 \sim 20$. The curves of the same color correspond to the same $d$ and are meant to be compared, solid lines correspond to the BBF method and dashed lines to the ANN method. Each point in the graph is a mean of 100 runs.