# Efficient algorithms to calculate the Hausdorff Distance on point sets represented by a $k^2$-tree

Fernando Domínguez[1], Gilberto Gutiérrez[1], Miguel R. Penabad[2*], Miguel Romero[1], Fernando Santolaya[1†]

[1]Department of Computer Science and Information Technology, Faculty of Economic Sciences, University of Bío-Bío, Chillán, Chile.
[2*]University of A Coruña, CITIC Research Center, A Coruña, Spain.

*Corresponding author(s). E-mail(s): miguel.penabad@udc.es;
Contributing authors: fdomingu@egresados.ubiobio.cl; ggutierr@ubiobio.cl; miguel.romero@ubiobio.cl; fsantolaya@ubiobio.cl;
†These authors contributed equally to this work.

**Abstract**

The Hausdorff distance is a measure of the similarity between two sets of points. It has been used in many different fields, such as comparing MRI images or transportation routes. There have been different approaches to compute the Hausdorff distance; some algorithms operate in main memory, while others store the set of points in secondary memory. In order to avoid secondary memory, compact data structures, such as $\boldsymbol{k^2}$-tree, can be used. They are able to index large sets of points in main memory, and they can be efficiently queried while minimizing storage. We present in this article two efficient algorithms (HDKP1 and HDKP2) to compute the Hausdorff distance over two data sets that are stored in $\boldsymbol{k^2}$-trees. These algorithms provide a time- and space-efficient solution. The performance of our algorithms was evaluated through a series of experiments together with the most promising algorithms from the state of the art. Based on the results, it was concluded that our approach is competitive or exceeds current algorithms.

**Keywords:** Algorithms, Compact Data Structures, Hausdorff distance, $\boldsymbol{k^2}$-tree

## 1 Introduction

The Hausdorff distance calculation is not a new problem and it is mostly addressed in the field of topology [1]. The first computer science study dates back to 1983 in which the authors [2] proposed an algorithm to calculate the Hausdorff distance of two convex polygons.

The Hausdorff distance is the similarity measure of two sets of points $A$ and $B$. Therefore, it can be used to compare geometric shapes, such as polylines, convex polygons [2], or general polyhedra represented as triangular meshes [3]. This has many practical applications because it is possible to model various situations in the real world as a set of points or as geometric figures. For example, this similarity measure is used in the field of topology to compare maps [1]. It is also used to compare images to identify the similarity between images of skin lesions from one patient to another in the field of dermatology [4], follow the evolution of the size of a brain tumor over time [4], and for facial recognition [5–7].

The $\mathsf{HausDist}(A, B)$ is a function where $A$ and $B \subseteq \mathbb{R}^d$ and is defined as [8]:

$$\mathsf{HausDist}(A, B) = \max_{a\in A}\{\min_{b\in B}\{\mathsf{Dist}(a, b)\}\}$$

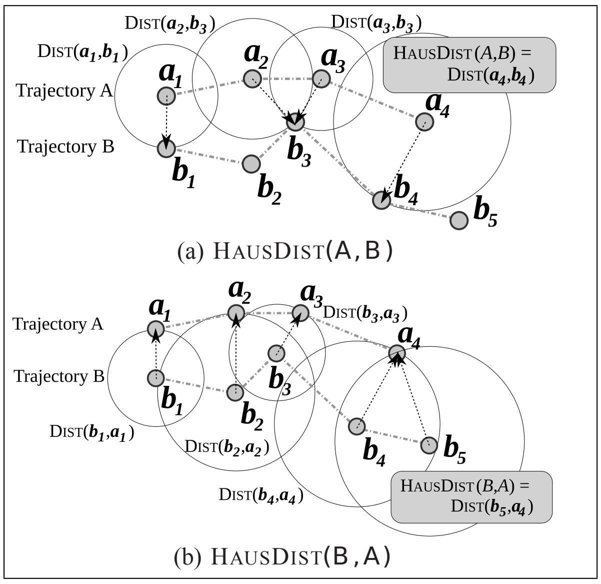
It is the greatest distance between each point $a \in A$ with its nearest neighbor $b \in B$. The $\mathsf{HausDist}(A, B)$ is also called the direct Hausdorff distance. The Hausdorff distance is not symmetric ($\mathsf{HausDist}(A, B) \neq \mathsf{HausDist}(B, A)$), so it is defined as [9]:

$$\mathsf{SymHD}(A, B) = \max\{\mathsf{HausDist}(A, B), \mathsf{HausDist}(B, A)\}$$

Fig. 1 illustrates two trajectories[1] $A$ and $B$. The Hausdorff distance can be regarded as the largest discrepancy between one trajectory and another; the notion of discrepancy between trajectories is a similarity measure [10]. Therefore, the largest discrepancy between $A$ and $B$ is $\mathsf{HausDist}(A, B) = \mathsf{Dist}(a_4, b_4)$ and the largest discrepancy between $B$ and $A$ is $\mathsf{HausDist}(B, A) = \mathsf{Dist}(b_5, a_4)$. A practical application of the aforementioned would be to identify the maximum additional distance a passenger would have to travel if the public transport authority decided to replace a bus route (sequence of stops) with another [9]. The different alternative routes could then evaluated to choose the one that generates the least impact on the population.

Given the increasing volume of information, it is indispensable to efficiently use storage to represent data. Data size not only has an impact on storage, but also on transmission and processing costs. One approach to address this problem comes from the field of information recovery systems, known as compact data structures [11]. These data structures efficiently represent information by using a space that is very close to the theoretical optimum according to information theory. They also maintain direct access properties to the data, which enables operations to be performed on the structure without previously decompacting it. In the previous examples, it is easy to visualize contexts in which great volumes of points exist, such as comparing high resolution images. These images are more costly to analyze when they require indexing

[1]A trajectory is the sequence of points through which a moving object has passed. The sequence of points is ordered by time.



(a) HAUSDIST(A, B)

(b) HAUSDIST(B, A)

**Fig. 1**: Example of Hausdorff distance calculation for both $A$ to $B$ and $B$ to $A$ [9].

the points in secondary memory than when they are in main memory, merely for the location of data. Various compact data structures to represent sets of points are encountered in the literature, among which are the wavelet tree [12] and the $k^2$-tree [13]. The wavelet tree is highlighted because it can obtain theoretical optimum storage as compared with the $k^2$-tree. However, a $k^2$-tree can achieve a lower storage cost when the points are clustered [11].

As far as we know, there are no algorithms that compute the Hausdorff distance using any compact data structures in the reviewed literature. Therefore, if the point sets are stored in one of them, there are basically two ways to calculate the Hausdorff distance. The first consists in extracting the points from the compact structure and then calculate the Hausdorff distance with some known algorithm. The second is to devise new algorithms to calculate the Hausdorff distance directly using the compact data structure. Two new algorithms are proposed in the present study[2], which were based on the ideas of [9], [8], and [15] to directly calculate the Hausdorff distance for sets of two-dimensional points stored in different $k^2$-trees.

Given that the $k^2$-tree is a static structure, the repeated calculation of the Hausdorff distance for the same sets is meaningless. It is more convenient to calculate it only once and store it because it does not vary. However, it is sometimes necessary to calculate the Hausdorff distance for subsets established by spatial ranges or for sets that are generated from other sets through set operations for $k^2$-trees, such as union and intersection, as described in [16].

The rest of the article is organized as follows. Section 2 reviews previous work on the Hausdorff distance problem and describes compact data structures, focusing on the

[2]A preliminary version of this work was presented at the CLEI International Conference [14].

$k^2$-tree, which will be used in our work. Section 3 describes our algorithms to calculate the Hausdorff distance. The experiments we conducted are shown in Section 4. Finally, the last section offers some conclusions and directions for future work.

## 2 Related Work and Background

In this section, we discuss the algorithms proposed in the literature to calculate the Hausdorff distance. Algorithms that assume it is possible to completely store two sets in main memory and those that process data from secondary memory are analyzed. Essential compact data structures are also described, and the $k^2$-tree structure used in the present study is highlighted.

### 2.1 Algorithms in main memory to compute Hausdorff distance

The first algorithm encountered in the literature to calculate the Hausdorff distance in main memory was proposed by Atallah [2]. It is used to calculate the Hausdorff distance between two disjoint convex polygons, and its temporal complexity is $\mathcal{O}(m+n)$ where $m$ and $n$ are the number of external vertices of each convex polygon. Following the same line of comparing polygonal shapes, Tang *et. al.* [17] demonstrated an algorithm to calculate the Hausdorff distance between complex polygonal models. Unlike the previous algorithm, it requires no special conditions for point sets. The proposed algorithm uses a Voronoi subdivision to divide the polygonal model into smaller triangles to calculate the Hausdorff distance between the vertices of those triangles. This algorithm only provides an approximate result. Helmut *et. al.* [18] presented algorithms that also use the Voronoi diagram to calculate Hausdorff distance with a temporal complexity of $\mathcal{O}((n+m)\log(n+m))$.

More recently, Taha *et. al.* [8] proposed an algorithm that eventually avoids calculating the nearest neighbor of each point of $A$ for the points of $B$.

The algorithm progressively optimizes (maximizes) an initial candidate solution in the following way. For each point $p$ of $A$, an attempt is made to calculate its nearest neighbor in $B$. If a point $q$ is obtained in $B$ in which the distance between $p$ and $q$ is less than the candidate solution reached so far, then the query for the nearest neighbor for $p$ is not continued (early break) and the next point of $A$ is selected.

The algorithm shows temporal complexity $\mathcal{O}(m)$ for its best case and $\mathcal{O}(mn)$ for its worst case.

Another algorithm that uses the *"early break"* technique has been proposed by Chen *et. al.* [19], which is known as a *local start search (LSS)*. The *"early break"* of this algorithm arises from the probability of high density zones of points. As in Taha *et. al.* [8], these *"early breaks"* are performed when calculating the nearest neighbor of a point. The algorithm uses a mechanism to record the position of the last break as an initial position to calculate the nearest neighbor of the next element of $A$. In another line of research, Guthe *et. al.* [20] submitted an algorithm to calculate the Hausdorff distance of geometric meshes. This algorithm takes advantage of the geometric mesh properties because it reviews only those regions where maximum distances exist between the triangles of each mesh. A grid was constructed using an octree data structure [21] to store each region by calculating the minimum and maximum distances between them.

Similarly, Kang *et. al.* [22] proposed an algorithm to calculate the Hausdorff distance between a triangular geometric mesh and a quadruple mesh. They tried to decrease the use of memory by eliminating the storage of all the combinations to avoid processing the whole set of information.

As for comparing tri-dimensional models and using the ideas proposed in [8], Zhang *et. al.* [23] submitted algorithms that use what the authors call *"diffusion search"* to optimize the calculation of the Hausdorff distance. They used an octree data structure as described in [20]. The main idea of these algorithms is to take advantage of the probability that if an *"early break"*, as defined in [8], the next solution is found in points near this first candidate solution. This idea is similar to the one described in [19]. Another interesting point about these algorithms is that they consider point distribution and density. When dealing with a dispersed set of points, the authors proposed an algorithm that reviews the points of the second z-order set. It converts these sets by using Morton code, which is a function that maps multidimensional data in a single dimension, thus preserving the location of the points [24].

Huttenlocher *et. al.* [25] proposed two algorithms to compare images; one is for objects in $\mathbb{R}^2$ with $\mathcal{O}(mn \log mn)$ complexity and the other for objects in $\mathbb{R}^3$ with $\mathcal{O}(m^2n^2 \propto (mn))$ complexity where $m$ and $n$ are the number of points of each set and $\propto (mn)$ is the translation function of these sets. In line with [2], these algorithms calculate the Hausdorff distance with the polygons generated by the exterior points of the data sets. When using this same translation method, but specifically for the calculation of the Hausdorff distance between two lines, Rote [26] and Li *et. al.* [27] proposed algorithms that improved the algorithm submitted in [25]. The complexity of these algorithms is $\mathcal{O}((n+m)\log(n+m))$. Depending on the number of points in each set, the problem with these translation techniques is that they require excessive computation time. To reduce it, Chang *et. al.* [28] proposed an algorithm that uses a Kalman filter [29] to filter points and improve computation time when comparing these sets.

More recently, Ryu and Kamata [30] proposed a new algorithm, based on the idea of ruling out points. It is also based on the early break technique, but in this case they apply it not only in the internal loop (as Taha's algorithm) but also in the external loop. Their time complexity is $\mathcal{O}(m)$ for its best case and $\mathcal{O}(mn)$ for its worst case, but their empirical tests (using 3D synthetic datasets and MRI images) show that they can outperform Taha [8] in up to 2 orders of magnitude.

The Hausdorff distance can also be calculated between curves. An algorithm for this purpose was presented by Chen *et. al.* [31]. This algorithm uses a heuristic method to divide the curves into sub-intervals through $B$-spline curves. It also uses geometric pruning techniques to eliminate those sub-intervals that do not contribute to the final solution.

## 2.2 The $k^2$-tree Compact Data Structure

Compact data structures enable space-efficient data representation, while still being able to efficiently solve required operations on the data [11, 13]. Compact data structures are available for binary sequences (*bitmaps*), symbol sequences, cardinal trees, permutations, graphs, and range indexing among others.

Among the compact data structures we can find the $k^2$-tree (see Fig. 2). Brisaboa *et. al.* [32] defined it as a compact data structure based on the quadtree [33]. This structure can also be regarded as a binary relation represented by a $k^2$-ary tree with height $h = \lceil \log_k n \rceil$; it represents a binary adjacency matrix with an $n \times n$ range in a compact manner. It can be used to represent general purpose graphs [34], RDF (resource description framework) data, or a bi-dimensional grid of points [35].

Fig. 2 shows a binary matrix and the conceptual $k^2$-tree (with $k = 2$) which stores the same information. The building process is as follows: starting with the whole matrix, it is divided in $k \times k$ equal-sized submatrices, and a bit is associated to each one. The bit is 1 if the corresponding matrix contains at least a 1, or a 0 if the whole submatrix is empty (full of 0s). The process continues recursively for all non-empty submatrices, and it stops when it either finds an empty submatrix or it reaches the individual cells of the matrix.

Finally, what is stored in the $k^2$-tree is the sequence of bits that corresponds to the breadth-first traversal of the conceptual tree, storing the last level (bitmap $L$) separately from the rest of the tree (bitmap $T$). For the example in Fig. 2, the stored bitmaps are $T = 1001$ and $L = 11011110$.

Space saving in a $k^2$-tree is due to the existence of quadrants that are full of 0s and are therefore not subdivided. In addition, the tree topology is not explicitly stored, saving the space required for the pointers and nodes of a classic tree representation.

The retrieval of direct and reverse neighbors, cell retrieval, and range queries are some operations that can be efficiently performed in a $k^2$-tree.

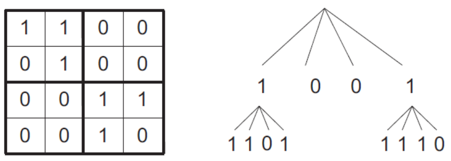


**Fig. 2**: Representation of a an adjacency matrix in a $k^2$-tree [32].

A variation of the $k^2$-tree is the hybrid approach, which tries to improve compression and query times. It maintains two $k$ values, a larger $k$ value for the upper levels and a smaller $k$ value for the lower levels. This reduces tree height.

Another special version of the $k^2$-tree uses the compression of 1s, which was originally described in [36]. This structure seeks to obtain an efficient representation for binary matrices that contain large groups of 0s and 1s. An example of these matrices are those representing binary images. The main difference with the original $k^2$-tree is the philosophy of submatrix division. The original decomposition in this structure stops when uniform regions of either 0s or 1s are found. This version has a higher compression ratio than the original $k^2$-tree when large regions of 1s appear in the binary matrix.

Another variation is the D$k^2$-tree, which is a dynamic version of the $k^2$-tree. The tree is modified when a cell of the adjacency matrix in this structure is dynamically changed. This enables subsequent modifications to the initial tree construction.

The storage required by a $k^2$-tree to store a set of points has been studied in different works [15, 35, 37]; these authors mainly conclude that the $k^2$-tree efficiently represents points, especially if the points are concentrated in regions of space.

## 2.3 Spatial indexes that can be used to speed up Hausdorff distance computation

It was our intention to compare the behavior of the $k^2$-tree with an spatial index. We are aware that R-trees [38] are one of the most well known spatial indexes. However, we do not consider them in this work, because they are basically designed to work well in secondary memory, and our data structures and algorithms are designed to work in main memory (RAM). For this reason, we decided to chose another well-known spatial index, the Kd-tree. This section briefly describes it, as well as a specific, compact version, the iKd-tree.

Let $P \subseteq \mathbb{R}^d$ be a set of $d$-dimensional points. A Kd-tree [39] is a data structure to store $P$, being also a multidimensional autoindex. It is a hierarchical structure (a binary tree) that recursively divides the space. Each node of the tree contains an object in $P$. Internal nodes divide the $d$-dimensional subspace in two subspaces by means of a $(d-1)$-dimensional hyperplane. This hyperplane is iso-oriented, and its direction alternates among the $d$ dimensions on the different levels of the binary tree. For example, if $d = 2$, the first partition could be based on the $X$ coordinate of the point stored in the root (level 0). The nodes on level 1 would partition the space according to the $Y$ coordinate, on level 2 according to $X$ again, and so on. The Kd-tree permits search, insertion and deletion operations on $O(h)$ time, being $h$ the height of the tree.

An iKd-tree [40] is an implicit Kd-tree that does not use pointers in its implementation. It is very convenient when the $P$ set is static. Basically, the iKd-tree stores the objects from $P$ in an array $Q[1\dots n]$, with $n = |P|$ that implicitly forms a balanced binary tree. The object stored in the root of the tree is located in $Q[\lceil \frac{n}{2} \rceil]$ and, according to the chosen hyperplane, divides the space in two subspaces like the original Kd-tree does. The left and right childs of the root (either internal nodes or leaves) are located at positions $Q[\lceil \frac{n}{4} \rceil]$ and $Q[\lceil \frac{3n}{4} \rceil]$, respectively. This way, exploring or navigating the iKd-tree is really simple.

As stated by [40], the cost of building a iKd-tree is $O(dn \log_2 n)$. It essentially corresponds to the cost of sorting the set $P$ according to each of the $d$ dimensions. The expected search time for a search (point query) is $O(\log_2 n)$.

## 3 Proposed algorithms

Our algorithms are based on ideas proposed by [8] and [41]. Specifically, the *"early break"* concept described by [8] is applied to both sets.

We shall use some metrics defined in [41] and [42], adapted to our algorithms. These works rely on the concept of MBRs (Minimum Bounding Rectangles) that enclose sets of points. However, in our algorithms, we do not really have MBRs, but "regions" that come from the recursive partition of a $k^2$-tree (or an iKd-tree, but since our major contribution is the design of algorithms that work with $k^2$-trees, we will focus on these). We define a *region* as either the whole binary matrix stored in the $k^2$-tree, or a submatrix that comes from the recursive partition of the $k^2$-tree.

Being $p_1$ and $p_2$ two points, and $R$ and $S$ two *regions*, the following functions are defined (see Fig. 3 and 5):

- $Dist(p_1, p_2)$ is the distance between $p_1$ and $p_2$. We consider the Euclidean distance, but other metrics, such as Manhattan or Chebyshev distances would also work.
- $minDist(p_1, S)$ is the minimum possible distance between $p_1$ and any point in $S$.
- $maxDist(p_1, S)$ is the maximum possible distance between $p_1$ and any point in $S$.
- $MAXMAXDIST(R, S)$ is the maximum possible distance from any point in $R$ to any point in $S$.
  Note that $maxDist(p_1, S)$ is a special case of *MAXMAXDIST(R,S)*, where $R$ is a region containing only the point $p_1$ [43].

Even when all these metrics were originally defined for MBRs or regions that represent rectangles, for our algorithms with $k^2$-trees, a region will always be a square, which comes from the recursive partition of the $k^2$-tree (if we consider also the iKd-trees, their recursive partition also obtains rectangles).

Also, for the sake of simplicity, we shall abuse the notation on some of those functions, by identifying a node in a $k^2$-tree with the region it identifies. Thus, for example, if $N$ is a node in the $k^2$-tree $K_A$ that represents the region $R$, we shall use interchangeably $minDist(p_1, N)$ and $minDist(p_1, R)$.

Our algorithms HDKP1 and HDKP2 are presented below. The first performs prunings only in set $B$, while the second applies pruning strategies in sets $A$ and $B$. A summary of the conditions and actions of the 4 pruning rules is shown in Table 1.

| **Rule** | **Condition** | **Action** |
|---|---|---|
| Rule 1 | $p \in A, pb \in B, Dist(p, pb) \leq cmax$ | Stop testing $p$ |
| Rule 2 | $p \in A, S \subseteq B, maxDist(p, S) \leq cmax$ | Stop testing $p$ |
| Rule 3 | $p \in A, S \subseteq B, minDist(p, S) > minNN$ | Do not test $S$ (do not insert in $pQ$) |
| Rule 4 | $R \subseteq A, S \subseteq B, MAXMAXDIST(R, S) \leq cmax$ | Do not test $R$ |

**Table 1**: Summary of pruning rules. $p$ and $pb$ are points, $R$ and $S$ are regions, $minNN$ is the current min distance from $p$ to its nearest neighbor (found so far).

## 3.1 HDKP1 algorithm

The algorithm assumes that the point sets $A$ and $B$ are stored in the $k^2$-trees $K_A$ and $K_B$, respectively. Basically, a $k^2$-tree represents a binary matrix (a matrix where each cell contains a 1 or a 0). So, a set of points is stored in a binary matrix simply setting to 1 the cell matrix $(x, y)$ for each point $(x, y)$ in the set, and setting to 0 all the remaining cells. Then, the matrix is stored in a compact form in the $k^2$-tree. For example, Fig. 4 shows on the left the matrix for the point set $\{(0, 4), (0, 7), (7, 7)\}$.

The HDKP1 algorithm extracts and explores each point in $K_A$ and, for each point, finds its nearest neighbor in $K_B$, provided that this neighbor is a possible solution.

The strategy of the algorithm is branching and pruning. It specifically uses three pruning rules based on the previously defined distance functions.

HDKP1 is shown in Algorithm 1. It heavily relies on the NNMax function (Algorithm 2). We will discuss in detail this function later, but basically NNMax obtains, for a point $p$ in $K_A$, its nearest neighbor in $K_B$.

In order to keep the pseudocode simple, we make some assumptions and use some specific notation. Consider we have a $k^2$-tree $K_A$. Then, $K_A$ also represents the root node of the conceptual tree. As we already mentioned, when we refer to any node $N$ in the conceptual $k^2$-tree, we also refer to the submatrix it represents. Thus, $maxDist(p, N)$ obtains the max possible distance between the point $p$ and any point in the submatrix the node $N$ represents. The following functions are also used in our pseudocode:

- $isLeaf(N)$ returns true if $N$ is a leaf node in a $k^2$-tree (it corresponds to an individual point), false otherwise.
- $hasChildren(N)$ returns whether an intermediate node $N$ in a $k^2$-tree has any children (whether it corresponds to a non-empty region in the matrix represented by the $k^2$-tree).

Basically, HDKP1 visits every point $p$ in the $k^2$-tree $K_A$ in a loop, and invokes NNMax (Algorithm 2) in each step. It obtains the nearest neighbor of the current $p$, and possibly improves the value of the Hausdorff distance (sometimes an iteration may skip the test using one of the pruning rules that we will discuss).

HDKP1 and NNMax share 3 important variables:

- $cmax$: it is the current candidate for the Hausdorff distance. It is initialized as 0.
- $pNN$: It is a global variable that contains the last computed nearest neighbor in $K_B$ to a point $p$ in $K_A$. It is initialized as $(\infty, \infty)$, and NNMax may change its value.
- $minNN$: It is the distance between the point $p \in K_A$ being processed, and $pNN$ (which, as we will see, is the nearest neighbor of the point $p$ computed in a previous iteration in the loop of HDKP1). It is used as a threshold that allows for some computations to be avoided (using pruning rules).

Let us describe HDKP1 following its pseudocode. The $for$ loop (lines 3-8) visits every point $p$ in $K_A$, and calls NNMax to find its nearest neighbor in $K_B$, with the aim of improving the value of $cmax$. Note that in line 5, NNMax is invoked only when the distance between $p$ and the current value of $pNN$ is higher than the current value of $cmax$. This is **pruning rule 1**: When this distance ($minNN$) is lower or equal to *cmax*, $p$ would not improve the answer, because we know there is a neighbor closer than the current *cmax* distance (we can consider $minNN$ an upper bound for the distance from $p$ to its nearest neighbor).

To clarify the application of pruning rule 1, we can refer to Fig. 3 in which, assuming that $p$ was processed before $q$, it is evident that if $minNN = Dist(q, pNN) \leq cmax$ is the distance between $q$ and $pNN$, it does not improve the value of $cmax$.

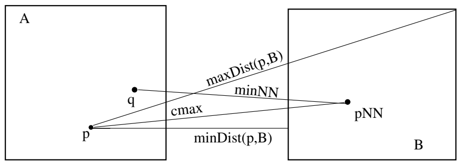


**Fig. 3**: Metrics used in pruning rules 1, 2, and 3.

Let us now focus on NNMax (Algorithm 2). It takes as imput the $k^2$-tree $K_B$, the current point $p$ being tested (from $K_A$), and the current values of *cmax* and $minNN$. The algorithm computes the nearest neighbor of $p$ in $K_B$, with the aim of improving the Hausdorff distance. It starts considering the whole $K_B$ and follows the $k^2$-tree partitioning schema to consider only subregions with points. Using pruning rules, NNMax can eventually avoid testing some subregions, or even stop calculating the nearest neighbor of the current $p$, deciding at an early stage that the distance between $p$ and its nearest neighbor does not improve the solution.

NNMax uses a priority queue represented by a min heap *pQ* that stores the regions of $K_B$ yet to be processed. Each *pQ* element has the $\langle n, d\rangle$ structure, where $n$ is a node of $K_B$, and $d$ is the maximum possible distance ($maxDist(p, n)$) between $p$ and the submatrix represented by $n$. *pQ* is sorted (in increasing order) by $d$, that is, the lowest value of $d$ is found at the top of the heap. This way, the regions with points close to $p$ are probably closer to the top of the queue and will be processed first.

This priority queue uses standard functions to access it. In our case, they are the following: *createMin-heap()* creates an empty min heap (priority queue), $empty(pQ)$ tests if *pQ* has no elements, $insert(pQ, element)$ inserts an element into *pQ*, and *extract-min(pQ)* extracts (and removes from the queue) the node at the top of the heap, which corresponds to the minimum distance.

NNMax initializes *pQ* (line 3) with an entry that represents the root of $K_B$, and the distance $maxDist(p, K_B)$. The algorithm is then executed in a cycle (lines 4-28), processing the *pQ* entries.

The $e$ entries of *pQ* which are intermediate nodes (not leaves) of $K_B$, are processed in lines 6-18. This process basically involves partitioning the node $e.n$ of $K_B$ (following the standard $k^2$-tree partitioning) and inserting the non-empty child nodes of $e.n$ in *pQ*. Note the optimization in this part of the algorithm, with the application of pruning rules 2 and 3, which will be defined now.

**Pruning rule 2** is applied in lines 7-9: If $e.d \leq cmax$ (being $e.d$ the maximum possible distance between $p$ and $e.n$, the currently dequeued node), then the nearest neighbor of $p$ in $K_B$ does not improve the solution. Therefore, no further exploration

**Algorithm 1 HDKP1** algorithm to calculate the Hausdorff distance for two point sets stored in $k^2$-tree.

```
    HDKP1($K_A$, $K_B$)
    input: Two $k^2$-trees $K_A$ and $K_B$.
    output: Hausdorff distance.
1: $pNN = (\infty, \infty)$          ▷ $pNN$ is a global variable that may be modified in NNMax
2: $cmax = 0$                                                  ▷ $cmax$ Hausdorff distance
3: for Each point $p$ in $K_A$ do
4:     $minNN = Dist(p, pNN)$
5:     if $minNN > cmax$ then                                   ▷ pruning rule 1
6:         $cmax =$ NNMax($K_B, p, cmax, minNN$)
7:     end if
8: end for
9: return $cmax$
```

of $K_B$ is necessary for the current $p$, so NNMax ends, returning the current value of $cmax$.

**Pruning rule 3** is applied in lines 13-16, before inserting an entry in $pQ$ . If $minDist(p, h)$, which is a lower bound of the distance of the nearest neighbor of $p$ in $h$, is not lower than $minNN$, which is the candidate distance between $p$ and its nearest neighbor at this point of the algorithm, child $h$ of the $e.n$ entry is not added to the priority queue $pQ$, because it does not minimize the value of $minNN$.

When the entries $e$ of $pQ$ being processed are leaves (they belong to the last level of $K_B$, and they represent points), they are processed in lines 19-26. Recall that only non-empty nodes are inserted in the priority queue, so in this type of entries, $e.n$ represents a point in $K_B$, and $e.d$ ($maxDist(p, e.n)$) is the real distance between points $p$ and $e.n$. In this case, **pruning rule 1** is also applied in NNMax, in lines 20-22: if the distance between $p$ and $e.n$ is not higher than $cmax$, then further exploration of $K_B$ is not necessary for the current $p$, because the value of $cmax$ would not improve, so the algorithm ends, returning the current value of $cmax$.

Finally, if the distance $e.d$ between $p$ and $e.n$ is lower than $minNN$ (lines 23-26), the algorithm updates $minNN$ and $pNN$.

As a final note on algorithms 1 and 2: as we mentioned, we have a global variable $pNN$, shared between them. It is initialized as $(\infty, \infty)$, but its value may be modified in Algorithm 2 (line 25), setting it to a point closer to the current $p$. Given the partitioning schema of the $k^2$-trees and the use of a min heap, it is very likely that the next point $p$ to be processed in Algorithm 1 would be close to the current one, obtaining a lower $minNN$, and thus increasing the probability of using the pruning rule 1 in HDKP1.

Now, let us use an example to walk through the algorithms HDKP1. The input is formed by two $16 \times 16$ binary matrices $A$ and $B$, which are shown in Fig. 4. They would be stored in two $k^2$-trees, $K_A$ and $K_B$.

Initially, $pNN = (\infty, \infty)$ (there is not yet any known point in $B$) and $cmax = 0$ (no current candidate; any result of $\mathsf{HausDist}(A, B)$ is $\geq 0$). Given the $k^2$-tree nature,

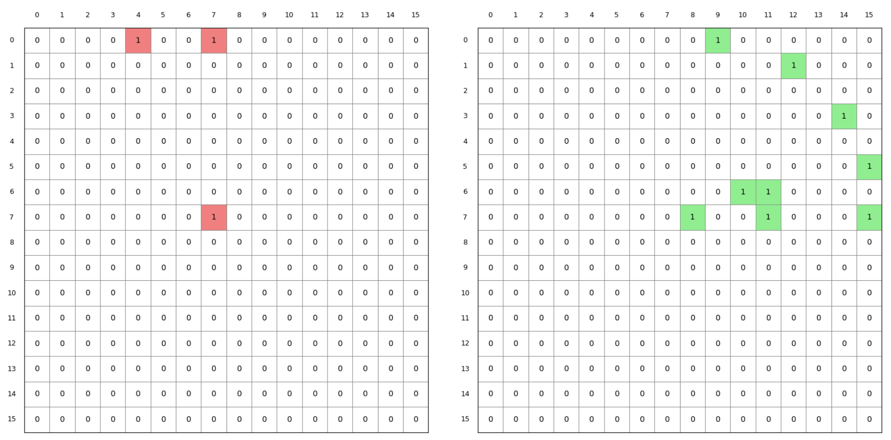
**Algorithm 2 NNMax.** Maximization of the Hausdorff distance.

```
NNMax($K_B, p, cmax, minNN$)
input: Root node of $k^2$-tree $K_B$, point $p \in K_A$, Hausdorff distance $cmax$ candidate, and $minNN$ the upper bound for the distance between $p$ and its nearest neighbor.
output: The updated Hausdorff distance $cmax$.
$pQ$ =createMin-heap( )
$d$ = maxDist($p, K_B$)
Insert($pQ, \langle K_B, d\rangle$)
while (Not Empty($pQ$)) do
    $e$ = Extract-Min($pQ$)
    if not isLeaf($e.n$) then
        if ($e.d \leq cmax$) then                          ▷ pruning rule 2
            return $cmax$
        end if
        for all Node $h$ child of $e.n$ do
            if ($hasChildren(h)$) then
                $hminDist$ = minDist($p, h$)
                if ($hminDist < minNN$) then               ▷ pruning rule 3
                    $hmaxDist$ = maxDist($p, h$)
                    Insert($pQ, \langle h, hmaxDist\rangle$)
                end if
            end if
        end for
    else
        if ($e.d \leq cmax$) then                          ▷ pruning rule 1
            return $cmax$
        end if
        if $e.d < minNN$ then
            $minNN = e.d$
            $pNN = e.n$                ▷ At leaf level, the submatrix is a point
        end if
    end if
end while
return $minNN$
```

$K_A$ will be explored using a depth first traversal, which corresponds to the $Z$-order. In our case, the points in $K_A$ will be processed in this order: $(0, 4)$, $(0, 7)$, $(7, 7)$.

For $p = (0, 4)$, $minNN = \infty$. The NNMax algorithm is invoked to compute the nearest neighbor of $p$ and eventually improve $cmax$. Additionally, it updates the global variable $pNN$, setting it to the closest point to $p$ in $B$.

What follows it the execution of NNMax for $p = (0, 4)$. $K_B$ is processed using the priority queue $pQ$ in order to visit the most promising quadrants. Specifically, it is initialized as $pQ = \{\langle((0, 0), (15, 15)), 346\rangle\}$, given that the region associated to



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 4**: Minimal example datasets for the running example.

the full $K_B$ is delimited by points$(0, 0)$ and $(15, 15)$ and $maxDist(p, (0, 0), (15, 15)) = |15 - 0|^2 + |15 - 4|^2 = 346$ [3].

Next, the while loop (lines 4 - 28) is executed. In the first iteration, $e = \{\langle((0, 0), (15, 15)), 346\rangle\}$. Since $e.n$ is not a leaf, it must be processed in lines 7-18. Rule 2 does not apply because $e.d > cmax$ $(346 > 0)$, thus all non empty children of $e.n$ must be processed (lines 10-18). In this case, only the quadrant $h = ((0, 8), (7, 15))$ is considered, because the other three do not contain any points. We compute $hminDist = minDist(p, h) = 16$. Since $hminDist < minNN$ $(16 < \infty)$, the entry $\langle((0, 8), (7, 15)), 170)$ is inserted in $pQ$, and the first iteration of the while loop ends.

The next iteration dequeues this last entry and processes it, being partitioned in 4 quadrants. Again, all of them are not leaves and no pruning rules apply, so they are inserted in $pQ$. Now, $pQ = \{\langle((0, 8), (3, 11)), 58)\rangle$, $\langle((4, 8), (7, 11)), 98)\rangle$, $\langle((0, 12), (3, 15)), 130)\rangle$, $\langle((4, 12), (7, 15)), 170)\rangle\}$.

The algorithm continues until the point $e.n = (0, 9)$ is extracted from $pQ$ and is processed in lines 19-26, where the global variables $minNN$ and $pNN$ are updated. At this point, the nearest neighbor for $p$ is $pNN = (0, 9)$ and the distance is $minNN = 25$. NNMax keeps running until $pQ$ is empty. In this example, both $pNN$ and $minNN$ remain unmodified. That is, the nearest neighbor of $p$ is the point $(0, 9)$ and the distance is 25. Finally, $minNN$ is returned, and this value updates $cmax$ in the algorithm HDKP1.

Again in HDKP1, the next point in $K_A$ to be processed is $p = (0, 7)$. Line 4 in HDKP1 computes $minNN = dist((0, 7), pNN) = 4$, and it applies pruning rule 1, since $minNN \leq cmax$ $(4 \leq 25)$, avoiding the computation of the nearest neighbor of $(0, 7)$, because it would not improve $cmax$.

The next point to be processed is $p = (7, 7)$. In this case, $minNN = Dist(p, pNN) = Dist((7, 7), (0, 9)) = 53$, so NNMax must be invoked.

[3]Note that, for efficiency purposes, and as defined in [42], the computed distance is the square of the Euclidean distance, skipping the final square root operation.

The entry $\langle((0,0),(15,15)),128\rangle$ is inserted in $pQ$. Processing it, the entry $\langle((0,8),(7,15)),113\rangle\}$ is inserted. Then, processing it, the priority queue becomes $pQ = \{\langle((0,8),(7,11)),25\rangle, \langle((0,8),(3,11)),85\rangle, \langle((0,12),(3,15)),113\rangle, \langle((4,12),(7,15)),154\rangle\}$. Processing the entry $\langle((0,8),(7,11)),25\rangle$, we can see that $e.d \leq cmax$ $(25 \leq 25\ )$. Therefore, pruning rule 2 is applied in line 7 of NNMax. This implies that the remaining entries in $pQ$ do not need to be processed, since the distance to nearest neighbor of $(7,7)$ will be at most 25, so it is not possible to improve $cmax$. At this point the algorithm HDKP1 ends. The answer is ) = 25. Or, more correctly, if we want to use Euclidean distances, $\sqrt{25} = 5$.

## 3.2 HDKP2 algorithm

The HDKP2 algorithm (Algorithm 3) is really an improvement of HDKP1, and it aims to avoid an exhaustive exploration of the points in both $K_A$ and $K_B$, in this case discarding (pruning) also whole regions of points of $K_A$. This is accomplished by using a new pruning rule (pruning rule 4), which is based on the metric *MAXMAXDIST* defined in [43] between quadrants or rectangles associated with the $k^2$-trees (see Fig. 5). The main idea for **pruning rule 4** is that, being $R$ and $S$ two regions in $K_A$ and $K_B$, respectively, *MAXMAXDIST(R, S)* represents the upper limit of the distance between any point in $R$ and any point in $S$. If $cmax$ is no lower than this upper limit, the whole region $R$ can be skipped, because it would not improve Hausdorff distance.

HKDP2 still uses NNMax (Algorithm 2) to apply the previously defined pruning rules 1-3.

The pseudocode for HDKP2 is shown in Algorithm 3. An important part of it is the *isCandidate* function (Algorithm 4) to obtain candidate areas to compute the Hausdorff distance, so we shall describe it first.

The *isCandidate* algorithm takes as its first argument a node of the $k^2$-tree $K_A$ ($nodeA$), which corresponds to a region of the $A$ dataset coming from the standard $k^2$-tree partitioning. The remaining arguments are the whole $K_B$ and the current candidate Hausdorff distance, $cmax$. The objective is to determine if $nodeA$ has candidate points to improve the solution. If there are no candidate points, *isCandidate* returns -1, otherwise it will return an upper bound for the distance to consider, as low as possible.

A valid upper limit for the distance would be $MAXMAXDIST(nodeA, K_B)$. However, it would be too high to be useful, so *isCandidate* tries to lower it. For example, consider the matrix on the left of Fig. 4: The MAXMAXDIST using the whole matrix would give a valid upper limit, but if we consider that there are points only in its first quadrant, the upper limit would be lower. For this process, the algorithm uses a min-heap $pR$ with elements of the form $\langle n, d\rangle$, where $n$ is a node coming from the recursive partitioning of $K_B$, and $d$ is a distance based on $MAXMAXDIST(nodeA, n)$. Being a min-heap, the lowest distance is located at the root of the heap. It is initialized (lines 1-3) with the whole $K_B$.

The $pR$ min-heap is processed in a loop (lines 4-19) to determine if $K_B$ contains candidate points that increase the value of $cmax$. If the node extracted from $pR$ is a leaf (line 11), that is, a point has been reached, this means that all the regions of $K_B$ have been scanned and it is impossible to apply pruning rule 4. Therefore, $R$ contains

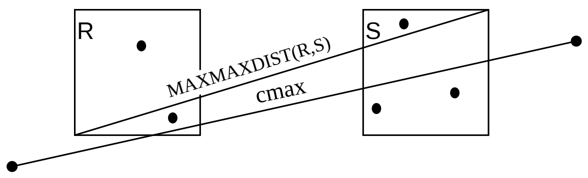


**Fig. 5**: Metric used in pruning rule 4. No points from R will improve *cmax* because they have neighbors in $S$ closer than *cmax*.

candidate points that could optimize *cmax*. In this case, the algorithm returns distance $d$ as an upper limit to the distance of the nearest neighbors of all points in *nodeA*.

It the extracted element $bb$ of $pR$ corresponds to a non-empty internal node, the algorithm processes its children nodes $h$ in lines 9-17, computing $maxDist = MAXMAXDIST(nodeA, bb.h)$ (trying to get a lower upper limit for the distance). Here, **pruning rule 4** can be applied: if *maxDist* is less than *cmax*, any nearest neighbor to the points in the region covered by *nodeA* has a smaller distance than *cmax*, so this region can be discarded, because it would not improve *cmax* (see, for example, Fig. 5). If *maxDist* is greater than *cmax*, the child node $h$ is enqueued in $pR$, along with *maxDist*.

Now let us describe $HDKP2$ (Algorithm 3). The initialization process (lines 1-5) includes computing an initial *cmax* candidate value (by getting the nearest neighbor of any random point from $K_A$) and the initialization of a priority queue $pQ$, which is implemented as a max-heap. Each element is of the form $\langle n, d\rangle$, where $n$ is a node in $K_A$ and $d$ is a distance computed by the *isCandidate* algorithm. This distance is initialized as $N \times \sqrt{2}$, which corresponds to $MAXMAXDIST(K_A, K_B)$, being the length of the diagonal of the matrix represented by $K_A$. $pQ$ is sorted in decreasing order of $d$, that is, the largest value of $d$ is found at the root of the heap.

Then the loop in lines 6-26 processes the entries of $pQ$. For each $aa$ node extracted in line 7, three scenarios may occur:

- Its distance is less than or equal to the current *cmax* (line 8). Since $pQ$ is a max-heap, it means that there are no nodes in it that may improve (increase) *cmax*, the the algorithm ends returning its current value.
- The $aa.n$ node is a leaf (line 11). In this case, the distance to its nearest neighbor in $K_B$ is computed using the previously described NNMax algorithm (Algorithm 2). If this value is larger than *cmax*, then *cmax* is updated. It is important to remember that NNMax uses pruning rules 2 and 3 to discard regions in $K_B$, optimizing the process.
- The $aa.n$ node is an internal node (line 17). In this case, each of its non-empty children are processed. The previously defined *isCandidate* algorithm (Algorithm 4) is used to determine if the region contains candidate points to improve *cmax* and, if so, the child in enqueued in $pQ$ along with the new upper limit for the distance.

**Algorithm 3 HDKP2** algorithm to calculate the Hausdorff distance for two point sets stored in $k^2$-tree, discarding quadrants in $K_A$.

```
HDKP2($K_A$, $K_B$)
input: Two $k^2$-trees $K_A$ and $K_B$.
output: The Hausdorff distance.
1:  $p$ = any point from$K_A$
2:  $cmax$ = NearestNeighbor($p$, $K_B$)
3:  pQ = CreateMax-Heap()
4:  $d = N \times \sqrt{2}$;                          ▷ highest value of cmax
5:  Insert($pQ, \langle K_A, d \rangle$)
6:  while (Not Empty($pQ$)) do
7:      $aa$ = Extract-Max($pQ$)
8:      if $aa.d \leq cmax$ then
9:          return $cmax$
10:     end if
11:     if isLeaf($aa.n$) then                     ▷ $aa.n$ is a point
12:         $nn$ = NNMax($K_B, aa.n, cmax, \infty$)
13:         if $nn > cmax$ then
14:             $cmax = nn$
15:         end if
16:     else
17:         for all Node $h$ child of $aa.n$ do
18:             if ($hasChildren(h)$) then
19:                 $dh$ = isCandidate($h, K_B, cmax$)
20:                 if ($dh \neq -1$) then
21:                     Insert($pQ, \langle h, dh \rangle$)
22:                 end if
23:             end if
24:         end for
25:     end if
26: end while
27: return $cmax$
```

The Hausdorff distance returned by HDKP2 is the current value of *cmax*, either returned in the early exit condition described in the first scenario, or when the max-heap *pQ* is exhausted.

As a final reminder, recall that Table 1 contains a summary of the conditions and actions of all 4 pruning rules.

## 3.3 Time complexity of our algorithms

A $k^2$-tree is a hierarchical data structure that is conceptually very similar to a Quadtree (specially when $k = 2$). Additionally, the (non simmetrical) Hausdorff distance between $A$ and $B$ can be modeled as a join of two sets, using $K_A$ and $K_B$,

**Algorithm 4 IsCandidate.** Determine if the quadrant of set A has candidate points for scanning.

```
IsCandidate($nodeA, K_B, cmax$)
input: Node (region) of $K_A$, $k^2$-tree $K_B$, the Hausdorff distance $cmax$ candidate.
output: $MAXMAXDIST(nodeA, S)$ for an $S \subseteq K_B$, or -1 if $nodA$ has no candidates to improve $cmax$.
1: $pR$ = CreateMin-Heap()
2: $d$ = MAXMAXDIST($nodeA, K_B$)
3: Insert($pR, \langle K_B, d\rangle$)
4: while (Not Empty($pR$)) do
5:     $bb$ = Extract-Min($pR$)
6:     if isLeaf($bb.n$) then
7:         return $bb.d$
8:     else
9:         for all Node $h$ child of $bb.n$ do
10:            if (hasChildren($h$)) then
11:                $maxDist$ = MAXMAXDIST($nodeA, h$)
12:                if ($maxDist \leq cmax$) then                ▷ pruning rule 4
13:                    return $-1$
14:                end if
15:                Insert($pR, \langle h, maxDist\rangle$)
16:            end if
17:        end for
18:    end if
19: end while
```

respectively. For this type of operation, using hierarchical data structures it is common to use models that predict the cost of processing the join operation. For example, the work [44] is integrally devoted to propose a model to estimate the cost of performing a spatial join considering data sets stored in R-trees (also a hierarchical data structure) and distance between objects as spatial predicates. The development of a cost model to predict the performance of our algorithms is by itself a challenge that is beyond the scope of this work. Therefore, we concentrate only in the performance of our algorithms for the worst case scenario. Since the worst case occurs when no pruning rules apply, the time complexity is basically the same for HDKP1 and HDKP2. In any case, we have included an empirical estimation of the average-case complexity in Section 4.4.5, following an approach to those used in [45] or [46]. It shows that our algorithms range from $O(n)$ to $O(n \log n)$.

Let $m_A$ and $m_B$ the number of points stored in the $k^2$-trees $K_A$ and $K_B$, respectively. Basically, the algorithm consists en finding, for each point $p_A \in K_A$, it nearest neighbor in $K_B$. This is done by invoking the NNMax algorithm, which is very similar to the algorithm introduced in [15] (except that in our case the variable $cmax$ may be updated). The worst case for NNMax happens when all the points in $K_B$ are at the same distance of $pA$ and $cmax$ is a small value that does not help pruning. According to [15], the time complexity of NNMax is $\mathcal{O}(m_B \log_2 m_B)$.

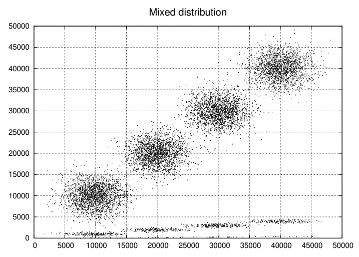


**Fig. 6**: Mixed distribution.

Our algorithms need to process $(\frac{n^2-1}{k^2-1})$ internal nodes, and $m_A$ leaves in $K_A$ (calling NNMax). Thus, the global complexity to solve $\mathsf{HausDist}(A, B)$ is $\frac{n^2-1}{k^2-1} + m_A + m_A(\mathcal{O}(m_B \log_2 m_B))$, which is $\mathcal{O}(m_A(m_B \log_2 m_B))$.

The complexity for the symmetrical Hausdorff distance, $\mathsf{SymHD}(A, B)$ would be, therefore, $\mathcal{O}(m_A(m_B \log_2 m_B) + m_B(m_A \log_2 m_A))$.

Of course, this worst case complexity is bad, almost quadratic on the number of points. However, the running time of our algorithms, as we will show in the following section, is much shorter. The algorithms benefit not only from the pruning rules, but also from the characteristics of the $k^2$-trees. For example, reaching an empty internal node in upper levels of the $k^2$-tree will decrease the number of internal nodes to be processed.

## 4 Experimental Evaluation

### 4.1 Experimental environment

The experiments were conducted on a computer with an Intel(R) Xeon(R) Silver 4309Y CPU @ 2.80GHz processor, 128 GB RAM Memory, and Ubuntu 22.04 LTS operating system. The performance of the experiments was measured by execution time and memory usage. The algorithms were implemented in the C++ programming language.

To conduct the experiments, it was assumed that the point sets are stored in two $k^2$-trees or two iKd-trees, depending on the version of the algorithms. Calculated mean time includes the time needed to extract the points from the data structures and the time required to calculate the symmetrical Hausdorff distance.

### 4.2 Data sets

#### 4.2.1 Synthetic data

Experiments were conducted with randomly generated point sets following the uniform and Gaussian distributions. Sets mixing both distributions were also considered (see Fig. 6). Points were generated in a two-dimensional space with a $2^{16} \times 2^{16}$ range. Set size was 10,000, 100,000, 1,000,000, and 10,000,000 points.

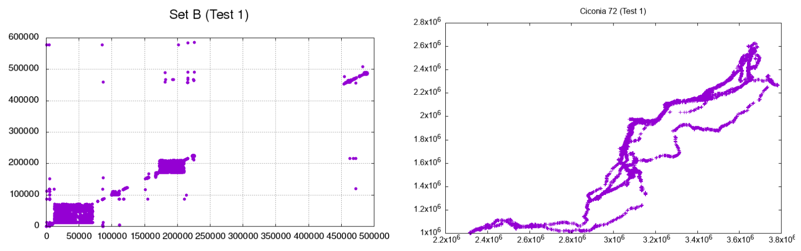


**Fig. 7**: Real data sets examples: LAW (left) and Ciconia (right).

To analyze execution time, six pairs of data sets were generated for each test, and the mean time used to calculate the Hausdorff distance for each algorithm was determined.

### 4.2.2 Real data

We used two data sets of real data. The first one represents sets of links between web pages of the `.uk` domain collected by the Laboratory for Web Algorithmics (LAW)[4]. Cells that are set to 1 (link between pages) in the adjacency matrix are considered as points of the set. The second dataset was obtained from [47], specifically the *Ciconia* dataset. It represents the trajectories of 88 white storks traveling between Europe and North Africa from 2013 to 2019. The points in this dataset represent GPS locations. The purpose of these data sets is to evaluate the performance of the algorithms against real distributions. Fig. 7 shows an example of the graphical distribution of real data with an example from each dataset. Set sizes for each test are displayed in Table 2.

| | # points $A$ | # points $B$ |
|---|---|---|
| Test 1 | 1,048,575 | 1,048,575 |
| Test 2 | 1,048,575 | 1,048,575 |
| Test 3 | 1,048,575 | 1,048,575 |
| Test 4 | 772,030 | 767,234 |

(a) LAW.

| | # points $A$ | # points $B$ |
|---|---|---|
| Test 1 | 210,275 | 214,551 |
| Test 2 | 210,275 | 259,331 |
| Test 3 | 210,275 | 227,355 |
| Test 4 | 214,551 | 259,331 |
| Test 5 | 214,551 | 227,355 |
| Test 6 | 259,331 | 227,355 |

(b) Ciconia.

**Table 2**: Number of points of datasets for the tests.

## 4.3 Tested algorithms and data structures

A series of experiments were conducted to evaluate the algorithms (HDKP1 and HDKP2) proposed in this work. In order to evaluate the performance of our algorithms, we compared them with two different proposals:

[4]http://law.di.unimi.it.

1. The algorithm proposed by Ryu and Kamata in [30]: It is, according to the literature, the most efficient option. The algorithm assumes that both sets of points are stored in arrays, without the support of any underlying data structure. It is designed for the symmetrical Hausdorff distance $\mathsf{SymHD}(A, B)$, and it greatly improves the second part, $\mathsf{HausDist}(B, A)$. We used our own implementation of their algorithm in our tests. Given that our algorithms compute only the direct Hausdorff distance, in our test we computed $\max(\mathsf{HausDist}(A, B), \mathsf{HausDist}(B, A))$ to obtain the same result.
2. iKd-tree. We have implemented an adaptation of HDKP1 and HDKP2 algorithms that operate directly over the iKd-tree data structure. In particular, pruning rules 1 to 4 were implemented, and the traversal of the tree considered the same strategies as the original algorithms over the $k^2$-tree.

We have made our code public, both for the $k^2$-tree[5] and iKd-tree[6] versions. Additionally, the datasets we used in our experiments, which will be described later in this seccion, are also publicly available[7].

## 4.4 Results

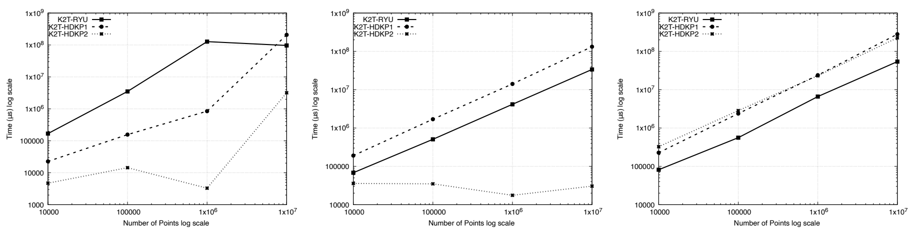
### 4.4.1 Comparison of algorithms HDKP1, HDKP2, and Ryu using synthtic data

First, we compared our algorithms, both HDKP1 and HDKP2, with Ryu and Kamata approach. Fig. 8 shows the results. We can notice that HDKP1 (K2T-HDKP1) and HDPK2 (K2T-HDPK2) outperform Ryu and Kamata (K2T-RYU) when they are applied to datasets with gaussian distribution (Fig. 8a). It is also evident that HDKP2 outperforms HDKP1 by 2 to 3 orders of magnitude, and Ryu and Kamata by 2 to 4. Considering a mixed distribution (Fig. 8b), HDPK2 is again faster than the alternatives. However, as Fig. 8c shows, Ryu and Kamata algorithm exceeds HDKP1 and HDKP2 (by about one order of magnitude) when the dataset distribution is random. The fact that HDKP2 has a better performance than Ryu and Kamata over the gaussian and mixed distributions is easily understandable: these distributions are more clustered, and this is a property that benefits the $k^2$-tree data structure. It also implies that the pruning rules, specifically pruning rule 4, will be more productive: a high level of clusterization implies that, when computing $\mathsf{HausDist}(A, B)$, if pruning rule 4 discards a submatrix of $A$, a higher number of points from $A$ are discarded.

### 4.4.2 Comparison of HKDP2 algorithms over $k^2$-tree and iKd-tree using synthtic data

We have also compared our algorithms with a different implementation of them, that is executed over datasets stored in an iKd-tree. Fig. 9 shows the performance of both versions in terms of execution time. We have considered only the HKDP2 algorithm, which was the fastest for both versions. It is easy to see that, over the gaussian

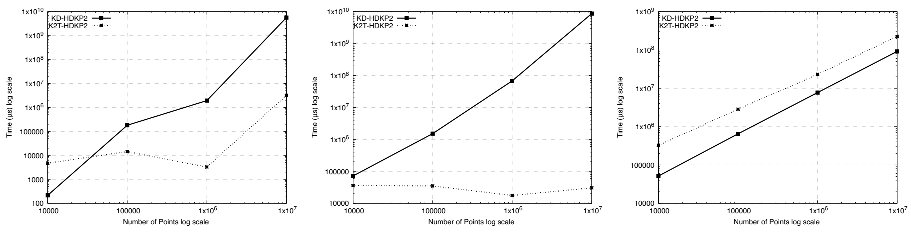
[5]https://github.com/fsantolaya/Hausdorff_k2tree
[6]https://github.com/fsantolaya/Hausdorff_kdtree
[7]https://doi.org/10.5281/zenodo.16258820



(a) Gaussian distribution. (b) Mixed distribution. (c) Random distribution.

**Fig. 8**: Execution time for K2T-HDKP1, K2T-HDKP2 y K2T-RYU algorithms considering synthetic data sets.

and mixed distributions, K2T-HDKP2 (the implementation of the algorithms over $k^2$-tree) outperforms, by several orders of magnitude, KD-HDKP (the implementation over iKd-tree). When the datasets grow, so does the difference in execution time, reaching up to 5 orders of magnitude for the larger datasets. However, although by a smaller difference, KD-HDKP2 outperforms K2T-HDKP2 for datasets with a random distribution. We can also see that this difference decreases as the dataset size increases. Like in Section 4.4.1, it is evident that the implementation over $k^2$-tree is clearly benefited by the data clustering and is hindered by random distributions.



(a) Gaussian distribution. (b) Mixed distribution. (c) Random distribution.

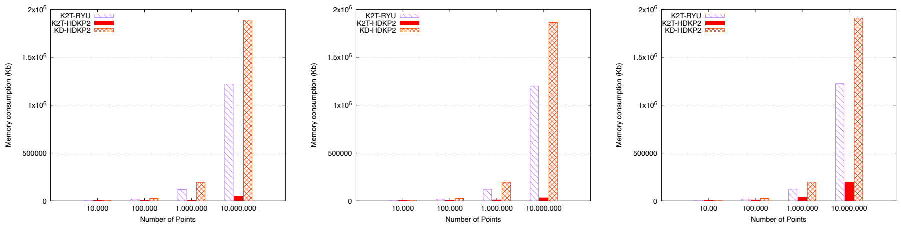
**Fig. 9**: Execution times algorithms HDKP2 using $k^2$-tree (K2T-HDKP2) and iKd-tree (KD-HDKP2).

### 4.4.3 Memory usage using synthetic data

Another important measure, apart from the running time of the algorithms, is the memory usage. We could have measured just the storage needed by the data structures, and the $k^2$-tree would be the one with the lowest storage footprint, because it is the only one that compresses the data. However, we believe that a more accurate measure is the peak memory usage of the programs that run the algorithms. In this way, we take into account the memory taken up by the data themselves, but also the additional structures needed to execute the algorithms. This is more realistic if we plan to run everything in RAM.

Fig. 10 shows the peak memory usage for our synthetic datasets. It is easy to see that the $k^2$-tree has a much lower memory demand in all cases and in all data distributions. In small datasets, the difference is less noticeable (around 80% of Ryu and iKd-tree versions). However, for large datasets, and inReal the best case (the mixed distribution), the $k^2$-tree memory demand can be as low as 2.72% (compared to Ryu) or even 1.75% (compared to iKd-tree). This means that, as we expected, the $k^2$-tree is able to deal with much larger datasets in main memory.

This difference in memory usage appears for several reasons. The first one is obvious: Ryu and Kd-trees deal with uncompressed data (arrays of points), while the $k^2$-tree uses compression. The second reason has to do with the use of priority queues (max-heaps and min-heaps). For $k^2$-trees and iKd-trees, the length of the priority queue is determined by the height of the tree, and it would be $\mathcal{O}(\log_{k^2} N)$ for $k^2$-trees, but $\mathcal{O}(\log_2 N)$ for iKd-trees, being $N$ the size of the original matrix. Given that, in our examples, we used $k = 2$, the height of the iKd-tree is twice the height of the $k^2$-tree. For the Ryu version, the extra memory comes from the use of two additional arrays of distances (each one associated to one of the datasets) of size $N$.



(a) Gaussian distribution (b) Mixed distribution. (c) Random distribution.

**Fig. 10**: Peak Memory Consumption of the three algorithms for synthetic data sets.

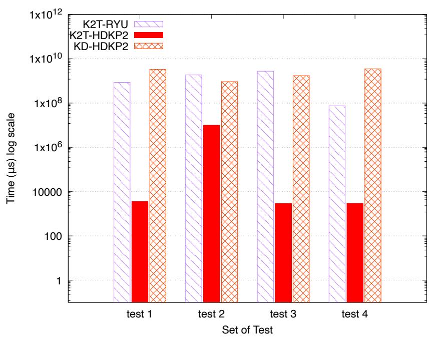
### 4.4.4 Real data

For real data sets, we decided to use again the most efficient algorithms and compare them. They are HDKP2 using a $k^2$-tree (K2T-HDKP2), using a iKD-tree (KD-HDKP2) and the Ryu and Kamata version using a $k^2$-tree(K2T-RYU). The execution times and memory usage are shown in Fig. 11 for the LAW dataset, and Fig. 12 for the Ciconia dataset.

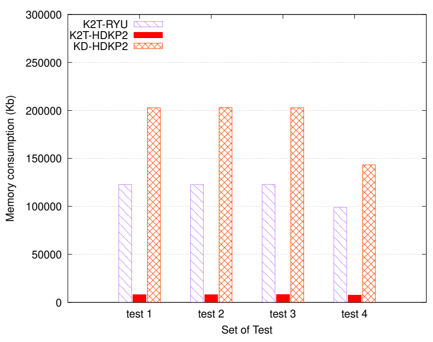
Let us focus first on the LAW dataset. Recall that the real data set is actually a web graph that represents links between pages in the `.uk` domain, and this type of graph has usually a high degree of clustering. Fig. 7 (left), graphically shows this clustering.

The execution times of the algorithms are consistent with those over the synthetic datasets that showed a higher degree of clustering (gaussian and mixed). Thus, K2T-HDKP2 shows the higher performance, being faster than its alternatives, by roughly 2 to 5 orders of magnitude. From Fig. 11a we can also conclude that the data used on test 2 is the less clustered, showing the lower difference among the execution times.

Regarding memory usage, we represent again the peak memory usage that represents all the memory used by the program, including the data sets and additional structures needed to run the algorithms. As Fig. 11b shows, the results are also consistent: the $k^2$-tree version has a much lower memory consumption. The Ryu and Kamata version may need up to 12 times more memory, and the iKd-tree up to 20 times.



(a) Execution time.



(b) Peak memory.

**Fig. 11**: Execution time and peak memory consumption of the algorithms over real data sets (LAW).

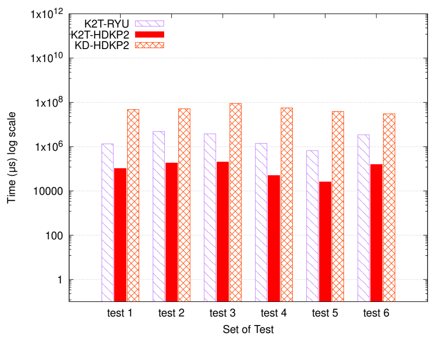
We can see the results for the Ciconia dataset in Fig. 12. Our results are consistent with the previous experiments: the K2T-HDKP2 algorithm outperforms the other approaches. It is easy to see that the difference is not so big as with the LAW dataset, but it is still better than Ryu's approach (between 1.3 and 2.9 orders of magnitude). This is probably due to the fact that the Ciconia dataset is not so clustered as LAW. Fig. 7 graphically shows it: while both graphics loosely resemble a diagonal of the matrix, the LAW datasets has clusters of points, while Ciconia contains something more similar to lines.

The same applies to memory consumption: it is always lower for K2T-HDKP2, but the difference with the other approaches is lower than with the LAW dataset (around 6-7 times less memory consumption than Ryu). This was expected, since the compression of the $k^2$-tree is highly dependent on the clustering degree.

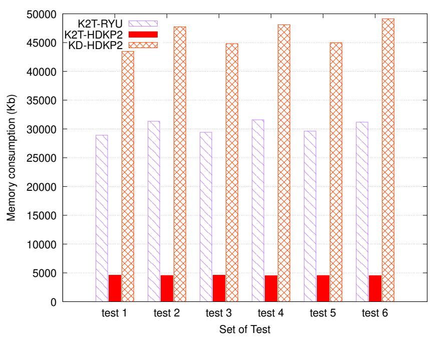
### 4.4.5 Empirical estimation of the time complexity for the average case

To estimate the algorithmic complexity for the average-case time, a nonlinear regression analysis was performed on the experimental data from the synthetic datasets. To develop the analysis, we used the function `curve_fit`, from the `scipy.optimize` module, implemented in Python.

From the experiments over synthetic datasets previously presented, for each combination of algorithm and data distribution, we have 30 observations. Each observation



(a) Execution time.



(b) Peak memory.

**Fig. 12**: Execution time and peak memory consumption of the algorithms over real data sets (Ciconia).

is the pair $(n, t)$, where $n$ is the number of points, and $t$ is the execution time of the algorithm.

Based on these observations, 11 different theoretical models of algorithmic complexity from the Big-$\mathcal{O}$ family were fitted. These models are of the form $T(n) = a \cdot f(n) + b$, where $a$ and $b$ are constants and $f(n)$ is one of the following: $\mathcal{O}(1)$, $\mathcal{O}(\log \log n)$, $O(\log n)$, $\mathcal{O}(\sqrt{n})$, $\mathcal{O}(n)$, $\mathcal{O}(n \log n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(n^2 \log n)$, $\mathcal{O}(n^3)$, $\mathcal{O}(n^3 \log n)$, $\mathcal{O}(n^4)$. After sorting (in descending order) the 11 obtained models based on their coefficient of determination ($R^2$), we selected the top 2 values, which represent the dominant empirical behavior of the algorithm over this data distribution. We decided to take the 2 greatest values of $R^2$ and not just the top 1 because their behavior was very similar, finding differences only at the third or fourth decimal digit, as Table 3 shows.

The model for each Big-$\mathcal{O}$ complexity class is the best among five different models obtained using the 5-fold cross-validation technique. This enables a more robust and generalizable estimation of the performance of the chosen model.

For the cross-validation, the 30 observations were divided in 5 subsets of similar size (folds). In each iteration, four sets were used to train the model and the remaining one to validate it. This process is repeated 5 times, generating 5 models.

During each validation, the model parameters were fitted using nonlinear least squares, with the coefficients constrained to non-negative values (because they represent computational costs). Then, the two models with higher coefficient of determination ($R^2$) were chosen. The performance measures of the selected model are the average $R^2$ and the average Mean Absolute Percentage Error (MAPE) across the five fitted models, rather than the values from the best model. Using the average provides a more robust estimate of the selected models $R^2$ and MAPE.

From the three studied data distributions, we observed that the one that better explains the behavior for the average-case is the Uniform distribution. Having the points uniformly distributed, they do not generate neither the best nor the worst case.

Table 3 presents the best two models for the algorithms, considering the data set with uniform distribution. As it shows, these two best models, that explain the asyntotic behavior for the average-case, are $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ for the three studied algorithms.

| **Top** | **Agorithm** | **Big O** | $R^2$ | **MAPE** | **Model** |
|---|---|---|---|---|---|
| 1 | K2T-RYU | $\mathcal{O}(n)$ | 0.9970 | 1.40% | $\hat{T}(n) = 5.37 \cdot n + 4.35 \cdot 10^5$ |
| 2 | K2T-RYU | $\mathcal{O}(n \log n)$ | 0.9962 | 2.40% | $\hat{T}(n) = 0.23 \cdot n \log n + 7.58 \cdot 10^5$ |
| 1 | K2T-HDKP2 | $\mathcal{O}(n)$ | 0.9969 | 0.30% | $\hat{T}(n) = 22.55 \cdot n + 4.50 \cdot 10^5$ |
| 2 | K2T-HDKP2 | $\mathcal{O}(n \log n)$ | 0.9966 | 1.29% | $\hat{T}(n) = 96.54 \cdot n \log n + 1.83 \cdot 10^{+6}$ |
| 1 | KD-HDKP2 | $\mathcal{O}(n \log n)$ | 0.9997 | 0.05% | $\hat{T}(n) = 0.39 \cdot n \log n + 2.07 \cdot 10^{-11}$ |
| 2 | KD-HDKP2 | $\mathcal{O}(n)$ | 0.9990 | 0.38% | $\hat{T}(n) = 9.15 \cdot n + 5 \cdot 10^{-16}$ |

**Table 3**: Top 2 estimated average-case complexity for each algorithm (Big O column), estimated Model and their associated $R^2$ and MAPE.

These models, obtained following the previously defined methodology, represent an empirical estimation of the complexity of the average-case for each analyzed algorithm. In particular, the estimations for the dataset with uniform distribution have a high $R^2$ and low MAPE. Thus, these results can be used as a working hypothesis to guide and support future theoretical studies on algorithmic cost and average-case analysis.

## 5 Conclusions

Two new algorithms have been presented in this work to calculate the Hausdorff distance between two point sets stored in $k^2$-trees. The algorithms can avoid processing all the points by using pruning rules. The first algorithm (HDKP1) applies the rules for the second set, and the second algorithm (HDKP2) applies the rules for both sets.

Through a series of experiments, our algorithms were compared with state of the art algorithm [30] using synthetic and real data. Based on 10,000 points and upward and using synthetic data, the experimental results showed that our algorithms outperformed the Ryu and Kamata version by (in the most favorable cases) up to 5 orders of magnitude in execution time. It also showed that the $k^2$-tree versions of the algorithms clearly benefit from clustered data sets. When using real data, these conclusions are confirmed, and the HDKP2 algorithm outperformed the Ryu and Kamata version by 1.3 to 5 orders of magnitude in all the studied cases.

Regarding memory consumption, our approach uses much less memory that its competitors, specially using large data sets. It ranges from 1.2 to 50 times less memory using synthetic data sets, and from 6 to 20 times using real data. Thus, we can conclude that our approach is able to handle much more large data sets in main memory.

According to the review of the state of the art and our own knowledge, this work is the first to provide an algorithm to calculate the Hausdorff distance when considering that the points are stored in a compact data structure (in this case, a $k^2$-tree).

As future work, we plan to extend our current algorithms, which work on 2 dimensions, to use a higher dimensionality. We also plan to explore possible optimizations on our current algorithms, such as the use of some cache with distances between some points or regions (similar to the work in [30]).

## Funding

Authors from University of Bío-Bío were supported in part by [grant numbers 2230534 IF/R, 2130520 IF/R] and by Agencia Nacional de Investigación y Desarrollo (ANID), FONDECYT 1230647 and 11251944. The author from University of A Coruña was partially supported by TED2021-129245B-C21 (PLAGEMIS): partially funded by MCIN/AEI/10.13039/501100011033 and NextGeneration EU/PRTR; PID2020-114635RB-I00 (EXTRACompact): partially funded by MCIN/ AEI/10.13039/501100011033; GRC: ED431C 2021/53, partially funded by GAIN/Xunta de Galicia; CITIC, as a center accredited for excellence co-financed by the EU through the FEDER Galicia 2021-27 operational program (Ref. ED431G 2023/01).

## References

[1] Kelley, J.L.: General Topology. Springer, New York, NY, USA (1975)

[2] Atallah, M.J.: A linear time algorithm for the Hausdorff distance between convex polygons. Information processing letters **17**(4), 207–209 (1983)

[3] Bartoň, M., Hanniel, I., Elber, G., Kim, M.-S.: Precise Hausdorff distance computation between polygonal meshes. Computer Aided Geometric Design **27**(8), 580–591 (2010)

[4] Spyridonos, P., Gaitanis, G., Bassukas, I.D., Tzaphlidou, M.: Gray Hausdorff distance measure for medical image comparison in dermatology: Evaluation of treatment effectiveness by image similarity. Skin Research and Technology **19**(1) (2013)

[5] Takacs, B.: Comparing face images using the modified Hausdorff distance. Pattern recognition **31**(12), 1873–1881 (1998)

[6] Gao, Y.: Efficiently comparing face images using a modified Hausdorff distance. IEE Proceedings-Vision, Image and Signal Processing **150**(6), 346–350 (2003)

[7] Jesorsky, O., Kirchberg, K.J., Frischholz, R.W.: Robust face detection using the Hausdorff distance. In: International Conference on Audio-and Video-Based Biometric Person Authentication, pp. 90–95 (2001). Springer

[8] Taha, A.A., Hanbury, A.: An efficient algorithm for calculating the exact Hausdorff distance. IEEE transactions on pattern analysis and machine intelligence **37**(11), 2153–2163 (2015)

[9] Nutanong, S., Jacox, E.H., Samet, H.: An incremental Hausdorff distance calculation algorithm. Proceedings of the VLDB Endowment **4**(8), 506–517 (2011)

[10] Trajcevski, G., Ding, H., Scheuermann, P., Tamassia, R., Vaccaro, D.: Dynamics-aware similarity of moving objects trajectories. In: Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems, p. 11 (2007). ACM

[11] Navarro, G.: Compact Data Structures: A Practical Approach, 1st edn. Cambridge University Press, New York, NY, USA (2016)

[12] Navarro, G.: Wavelet trees for all. In: Annual Symposium on Combinatorial Pattern Matching, pp. 2–26 (2012). Springer

[13] Ladra González, S.: Algorithms and compressed data structures for information retrieval. PhD thesis, Universidade da Coruna (2011)

[14] Domínguez, F., Gutierrez, G., Romero, M.: Algorithm to calculate the Hausdorff Distance on sets of points represented by k2-tree. In: 2018 XLIV Latin American Computer Conference (CLEI), pp. 482–489 (2018). https://doi.org/10.1109/CLEI.2018.00064

[15] Santolaya, F., Caniupán, M., Gajardo, L., Romero, M., Torres-Avilés, R.: Efficient computation of spatial queries over points stored in k2-tree compact data structures. Theoretical Computer Science **892**, 108–131 (2021) https://doi.org/10.1016/j.tcs.2021.09.012

[16] Quijada-Fuentes, C., Penabad, M.R., Ladra, S., Gutiérrez, G.: Set operations over compressed binary relations. Information Systems **80**, 76–90 (2019)

[17] Tang, M., Lee, M., Kim, Y.J.: Interactive Hausdorff distance computation for general polygonal models. ACM Transactions on Graphics (TOG) **28**(3), 74 (2009)

[18] Alt, H., Behrends, B., Blömer, J.: Approximate matching of polygonal shapes. In: Proceedings of the Seventh Annual Symposium on Computational Geometry, pp. 186–193 (1991). ACM

[19] Chen, Y., He, F., Wu, Y., Hou, N.: A local start search algorithm to compute exact Hausdorff distance for arbitrary point sets. Pattern Recognition **67**, 139–148 (2017)

[20] Guthe, M., Borodin, P., Klein, R.: Fast and accurate Hausdorff distance calculation between meshes. Journal of WSCG **13**(2), 41–48 (2005)

[21] Meagher, D.: Geometric modeling using octree encoding. Computer graphics and image processing **19**(2), 129–147 (1982)

[22] Kang, Y., Kyung, M.-H., Yoon, S.-H., Kim, M.-S.: Fast and robust Hausdorff distance computation from triangle mesh to quad mesh in near-zero cases. Computer Aided Geometric Design **62**, 91–103 (2018)

[23] Zhang, D., Zou, L., Chen, Y., He, F.: Efficient and accurate Hausdorff distance computation based on diffusion search. IEEE Access **6**, 1350–1361 (2018)

[24] Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd, Ottawa, Canada (1966)

[25] Huttenlocher, D.P., Kedem, K.: Computing the minimum Hausdorff distance for point sets under translation. In: Proceedings of the Sixth Annual Symposium on Computational Geometry, pp. 340–349 (1990). ACM

[26] Rote, G.: Computing the minimum Hausdorff distance between two point sets on a line under translation. Information Processing Letters **38**(3), 123–127 (1991)

[27] Li, B., Shen, Y., Li, B.: A new algorithm for computing the minimum Hausdorff distance between two point sets on a line under translation. Information Processing Letters **106**(2), 52–58 (2008)

[28] Chang, F., Chen, Z., Wang, W., Wang, L.: The Hausdorff distance template matching algorithm based on Kalman filter for target tracking. In: Automation and Logistics, 2009. ICAL'09. IEEE International Conference On, pp. 836–840 (2009). IEEE

[29] Kalman, R.E.: A new approach to linear filtering and prediction problems. Journal of basic Engineering **82**(1), 35–45 (1960)

[30] Ryu, J., Kamata, S.-i.: An efficient computational algorithm for Hausdorff distance based on points-ruling-out and systematic random sampling. Pattern Recognition **114**, 107857 (2021) https://doi.org/10.1016/j.patcog.2021.107857

[31] Chen, X.-D., Ma, W., Xu, G., Paul, J.-C.: Computing the Hausdorff distance between two B-spline curves. Computer-Aided Design **42**(12), 1197–1206 (2010)

[32] Brisaboa, N.R., Ladra, S., Navarro, G.: k2-trees for compact web graph representation. In: International Symposium on String Processing and Information Retrieval, pp. 18–30 (2009). Springer

[33] Samet, H.: The quadtree and related hierarchical data structures. ACM Computing Surveys (CSUR) **16**(2), 187–260 (1984)

[34] Álvarez-García, S., Freire C., B., Ladra, S., Pedreira, O.: Compact and efficient representation of general graph databases. Knowledge and Information Systems, 1–32 (2018)

[35] Bernardo Roca, G.: New data structures and algorithms for the efficient management of large spatial datasets. PhD thesis, Universidade da Coruna (2014)

[36] Bernardo, G., Álvarez-García, S., Brisaboa, N.R., Navarro, G., Pedreira, O.: Compact querieable representations of raster data. In: Procs. of SPIRE, pp. 96–108 (2013)

[37] Brisaboa, N.R., Ladra, S., Navarro, G.: Compact representation of web graphs with extended functionality. Information Systems, 152–174 (2014)

[38] Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD 1984, pp. 47–57 (1984). https://doi.org/10.1145/602259.602266

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

[40] Brown, R.A.: Building a balanced $k$-d tree in $o(kn \log n)$ time. Journal of Computer Graphics Techniques (JCGT) **4**(1), 50–68 (2015)

[41] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Closest pair queries in spatial databases. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. SIGMOD '00, pp. 189–200. ACM, New York, NY, USA (2000). https://doi.org/10.1145/342009.335414

[42] Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. SIGMOD Rec. **24**(2), 71–79 (1995) https://doi.org/10.1145/568271.223794

[43] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Closest pair queries in spatial databases. SIGMOD Rec. **29**(2), 189–200 (2000) https://doi.org/10.1145/335191.335414

[44] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Cost models for distance joins queries using R-trees. Data & Knowledge Engineering **57**(1), 1–36 (2006)

[45] Agenis-Nevers, M., Bokde, N.D., Yaseen, Z.M., Shende, M.: An empirical estimation for time and memory algorithm complexities: Newly developed R package. arXiv preprint arXiv:1911.01420 (2020)

[46] Goldsmith, S.F., Aiken, A.S., Wilkerson, D.S.: Measuring empirical computational complexity. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ESEC-FSE '07, pp. 395–404. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1287624.1287681 . https://doi.org/10.1145/1287624.1287681

[47] Gómez Brandón, A.: Object Trajectories. figshare (2021). https://doi.org/10.6084/m9.figshare.c.5740388.v2 . https://figshare.com/articles/collection/Object_Trajectories/5740388/2