Approximating the Directed Hausdorff Distance

Oliver A. Chubet ✉
Department of Computer Science, North Carolina State University

Parth M. Parikh ✉
Department of Computer Science, North Carolina State University

Donald R. Sheehy ✉
Department of Computer Science, North Carolina State University

Siddharth S. Sheth ✉
Department of Computer Science, North Carolina State University

──── **Abstract** ────

The Hausdorff distance is a metric commonly used to compute the set similarity of geometric sets. For sets containing a total of n points, the exact distance can be computed naïvely in $O(n^2)$ time. In this paper, we show how to preprocess point sets individually so that the Hausdorff distance of any pair can then be approximated in linear time. We assume that the metric is doubling. The preprocessing time for each set is $O(n \log \Delta)$ where Δ is the ratio of the largest to smallest pairwise distances of the input. In theory, this can be reduced to $O(n \log n)$ time using a much more complicated algorithm. We compute $(1 + \varepsilon)$-approximate Hausdorff distance in $(2 + \frac{1}{\varepsilon})^{O(d)} n$ time in a metric space with doubling dimension d. The k-partial Hausdorff distance ignores k outliers to increase stability. Additionally, we give a linear-time algorithm to compute directed k-partial Hausdorff distance for all values of k at once with no change to the preprocessing.

1 Introduction

The Hausdorff distance is a metric on compact subsets of a metric space. Let $(X, \mathbf{d})$ be a metric space and let A and B compact subsets of X. The distance from a point $x \in X$ to the set B is $\mathbf{d}(x, B) := \min_{b \in B} \mathbf{d}(x, b)$. The *directed Hausdorff distance* is $\mathbf{d}_h(A, B) := \max_{a \in A} \mathbf{d}(a, B)$, and the (undirected) *Hausdorff distance* is $\mathbf{d}_H(A, B) := \max\{\mathbf{d}_h(A, B), \mathbf{d}_h(B, A)\}$. This definition leads directly to a quadratic time algorithm for finite sets.

In our approach, we first preprocess the input sets A and B individually into linear-size metric trees, specifically, greedy trees [4]. All the points of a set are stored as leaves in the greedy tree and an internal node approximates the leaves in its subtree by a ball. A subset of greedy tree nodes of radius at most ε such that every point of the set is a leaf of exactly one node forms an ε-net of the underlying set. The greedy tree can be used to construct ε-nets of the underlying set at different scales. Our algorithms use greedy trees to maintain such nets for both sets and tracks which nodes of B are close to nodes of A. This approach batches the searches of the naïve algorithm and results in fewer distance computations.

If the input contains a total of n points, then preprocessing takes $\left(\frac{1}{\varepsilon}\right)^{O(d)} n \log \Delta$ time. Here Δ is the *spread* of the input sets, and d is the doubling dimension of the metric space. We present an algorithm that computes a $(1 + \varepsilon)$-approximation of the directed Hausdorff distance from A to B in $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)} n$ time after preprocessing. The preprocessing is especially useful when the same sets are involved in multiple distance computations.

One difficulty of working with the Hausdorff distance is its sensitivity to outliers. There are several variations of the Hausdorff distance that reduce the sensitivity to outliers. Among the simplest is the following definition where one can ignore up to k outliers. The kth*-partial*

directed Hausdorff distance is $\mathbf{d}_h^{(k)}(A, B) := \min_{S \in A^{(k)}} \mathbf{d}_h(S, B)$ where $A^{(k)}$ is the set of all subsets of A with k points removed [10].

One might expect that this is a harder problem than computing the directed Hausdorff distance. However, we show that for approximations in low dimensions, this is not the case; the worst-case running time matches that of our algorithm for the standard case (up to an additive $\log \Delta$ term). We present an algorithm that computes a $(1 + \varepsilon)$-approximation of $\mathbf{d}_h^k(A, B)$ for all values of k in $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)} n + O(\log \Delta)$ time after the same preprocessing as before. That is, the output is a list of $n + 1$ approximate values of $\mathbf{d}_h^k(A, B)$ for $k \in \{0, \dots, n\}$ and this list is produced in linear time.

2   Related work

We focus on general metrics with bounded doubling dimension. In the general setting, one may not expect a subquadratic algorithm for computing the Hausdorff distance, however, there are classes of metric spaces for which the Hausdorff distance can be computed more quickly. For example, given point sets in the plane, fast nearest neighbor search data structures [1] are used to give an $O(n \log n)$ time algorithm. If one allows for approximate answers, $O(n \log n)$ time algorithms are possible in low-dimensional Euclidean spaces [2].

It is not easy to get an asymptotic improvement on the naïve Hausdorff distance algorithm without using an efficient data structure in higher dimensions. In practice, there exist many heuristics to speed up the naïve algorithm [3, 15, 18]. Another popular technique is to use a geometric tree data structure. Zhang et al. [19] use octrees to compute the exact Hausdorff distance between 3D point sets. Nutanong et al. [12] present an algorithm to compute the exact Hausdorff distance using R-trees.

The partial Hausdorff distance was first introduced by Huttenlocher et al. [10]. Although there has been considerable interest in this pseudometric, most results are experimental and to the best of our knowledge, a theoretical running time bound does not exist. We give an algorithm to compute approximate partial Hausdorff distance that runs in linear time after preprocessing.

The algorithms presented in this paper maintain both input sets as metric trees and traverse them simultaneously. This approach is similar to that of Nutanong et al. [12], but more generally it has been studied in the machine learning literature as dual-tree algorithms [8]. Search problems such as the k-nearest-neighbor search [5], range search [5], and the all-nearest-neighbor search [14] have been explored using dual-tree algorithms. Any all-nearest-neighbor search algorithm where the query and reference sets are different can also be used to compute the directed Hausdorff distance. Ram et al. [14] present an all-nearest-neighbor algorithm that runs in linear time under some strict assumptions about the underlying metrics. The running time of their algorithm depends on a constant called the degree of bichromaticity. Moreover, this dependence is exponential in the degree of bichromaticity, and high values can result in a poor bound [6].

3   Background

3.1   Doubling metrics

Let $(X, \mathbf{d})$ be a *metric space*. A *metric ball* is a subset $\mathbf{ball}(c, r)$ of X, with center $c \in X$ and radius $r \geq 0$ such that $\mathbf{ball}(c, r) := \{x \in X \mid \mathbf{d}(x, c) \leq r\}$. The *spread* Δ of $A \subseteq X$ is the ratio of the diameter to the smallest pairwise distance of points in A. The *doubling dimension* of X, denoted $\dim(X)$, is the smallest real number d such that any metric ball in

X can be covered by at most 2^d balls of half the radius. If $\dim(X)$ is bounded then X is a *doubling metric*. The set A is *r-packed* if $\mathbf{d}(a, b) \geq r$ for all distinct $a, b \in A$. The following lemma by Krauthgamer and Lee [11] limits the cardinality of packed and bounded sets.

▶ **Lemma 1** (Standard Packing Lemma). *Let* $(X, \mathbf{d})$ *be a metric space with* $\dim(X) = d$. *If* $Z \subseteq X$ *is r-packed and can be covered by a metric ball of radius* R *then* $|Z| \leq \left(\frac{4R}{r}\right)^d$.

3.2 Greedy permutations and predecessor maps

Let $P = (p_0, \dots, p_{n-1})$ be a finite sequence of points in a metric space. The ith-*prefix* is the set $P_i = \{p_0, \dots, p_{i-1}\}$ containing the first i points of P.

The sequence P is a *greedy permutation* if for all i,

$$\mathbf{d}(p_i, P_i) = \max_{p \in P} \mathbf{d}(p, P_i).$$

For a constant $\alpha > 1$, the sequence P is an α-*approximate greedy permutation* if for all i,

$$\mathbf{d}(p_i, P_i) \geq \frac{1}{\alpha} \max_{p \in P} \mathbf{d}(p, P_i).$$

The point $p_0 \in P$ is called the *root* of the greedy permutation.

The *predecessor mapping* $T : P \setminus \{p_0\} \to P$ maps each non-root point p_i in P to an approximate nearest neighbor in the prefix P_i. The approximate nearest predecessor of a point need not be unique, however for the sake of construction, we assume we have chosen one. The *insertion distance* of a point $p \in P$, denoted ε_p, is defined as,

$$\varepsilon_p = \begin{cases} \mathbf{d}(p, T(p)) & p \neq p_0 \\ \infty & p = p_0. \end{cases}$$

An α-*scaling* predecessor map is one where every point has an insertion distance that is at most $1/\alpha$-times the predecessor's insertion distance, i.e., for every $p \in P \setminus \{p_0\}$,

$$\varepsilon_p \leq \frac{1}{\alpha} \varepsilon_{T(p)}.$$

We will later show how a greedy permutation and a predecessor map can be used to build a hierarchical search data structure.

Constructing greedy permutations

We present a simple algorithm to construct α-approximate greedy permutations. The input is a point set in a metric space and an optional root point. If no root is specified, then an arbitrary point can be chosen as the root. The algorithm constructs the permutation iteratively while simultaneously constructing an α-scaling predecessor map. Each uninserted point keeps track of its α-approximate nearest neighbor among the inserted points. This point is called the *parent* of the uninserted point.

We initialize the output permutation with only the root as an inserted point and all other points have the root as the parent. In each iteration, the uninserted point p maximizing the distance to its parent is inserted. The predecessor map is updated to store the parent of p as its predecessor. For every uninserted point, update its parent to p if p is closer than its current parent by a factor of α. We refer to these parent updates as α-lazy updates. An example of this algorithm is shown in Figure 1.
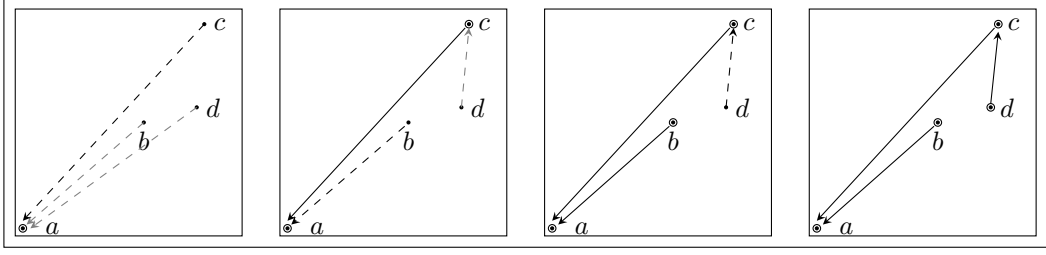
Figure 1 In this figure we compute the greedy permutation and predecessor mapping on a finite metric space $(\{a, b, c, d\}, \mathbf{d})$ with seed a and $\alpha = 2$. The dotted lines indicate parents. The darker dotted line highlights the point furthest from its parent. The solid lines indicate predecessors. Initially only a is inserted and it is the parent of every uninserted point. When c is inserted, it becomes the new parent of d because $\alpha \mathbf{d}(c, d) < \mathbf{d}(a, d)$. The parent of b is still a even though c is closer. Then, the insertion of b does not change the parent of d because c is still an α-approximate nearest neighbor. The completed permutation is (a, c, b, d) and the predecessor map is $b \mapsto a, c \mapsto a, d \mapsto c$.

In each iteration, the inserted point is α-approximately the farthest point and so the algorithm constructs an α-approximate greedy permutation. It is easy to see that if $\alpha = 1$ then the output is an exact greedy permutation. The α-lazy updates guarantee that every point has a distance to its parent that is at most $1/\alpha$ times that parent's insertion distance. So, it follows that the insertion distance of a point is at most $1/\alpha$-times that of its predecessor. Therefore, the constructed predecessor map is α-scaling.

The algorithm presented above was analyzed by Gonzalez [7] in 1985 and it runs in $O(n^2)$ time. More efficient algorithms can construct greedy permutations in $O(n \log \Delta)$ time for low-dimensional data [9, 16]. There also exists a randomized algorithm that can compute greedy permutations in $O(n \log n)$ time [9].

3.3 Greedy trees

A balltree [13] is defined by recursively partitioning compact subsets of a metric space and representing the partitions in a binary tree. For a ball tree on A, every node x has a center $\mathsf{ctr}(x) \in A$, and a radius $\mathsf{rad}(x)$. The set of all points contained in the leaves of the subtree rooted at node x is denoted by $\mathsf{pts}(x)$. The radius $\mathsf{rad}(x)$ is such that $\mathsf{pts}(x) \subseteq \mathbf{ball}(\mathsf{ctr}(x), \mathsf{rad}(x))$. Two nodes x and y are *independent* if $\mathsf{pts}(x) \cap \mathsf{pts}(y) = \varnothing$. We denote $\mathbf{d}(\mathsf{ctr}(x), \mathsf{ctr}(y))$ as $\mathbf{d}_{\mathsf{ctr}}(x, y)$.

A greedy tree [4] is a ball tree that can be built on A using the greedy permutation on A and a predecessor map on this permutation. Every node has two children such that the left child is centered at the same point as the parent. The root of the greedy tree is centered at the first point of the greedy permutation. The rest of the tree is constructed incrementally. At all times, there is a unique leaf centered at each of the previously inserted points. For every point b in the permutation, let a denote its predecessor. Create two nodes, one centered at a and one centered at b. Attach these nodes as the left and right children respectively of the leaf centered at a (see Figure 2).

The radius of a node is the distance from the center to the farthest leaf in its subtree. Thus, leaves have a radius of zero. The radii of all other nodes can be computed by traversing the subtrees. The radius of a child is never greater than the radius of its parent. If the greedy permutation is α-approximate and the predecessor map is α-scaling, then the resulting greedy tree is said to have a parameter α.
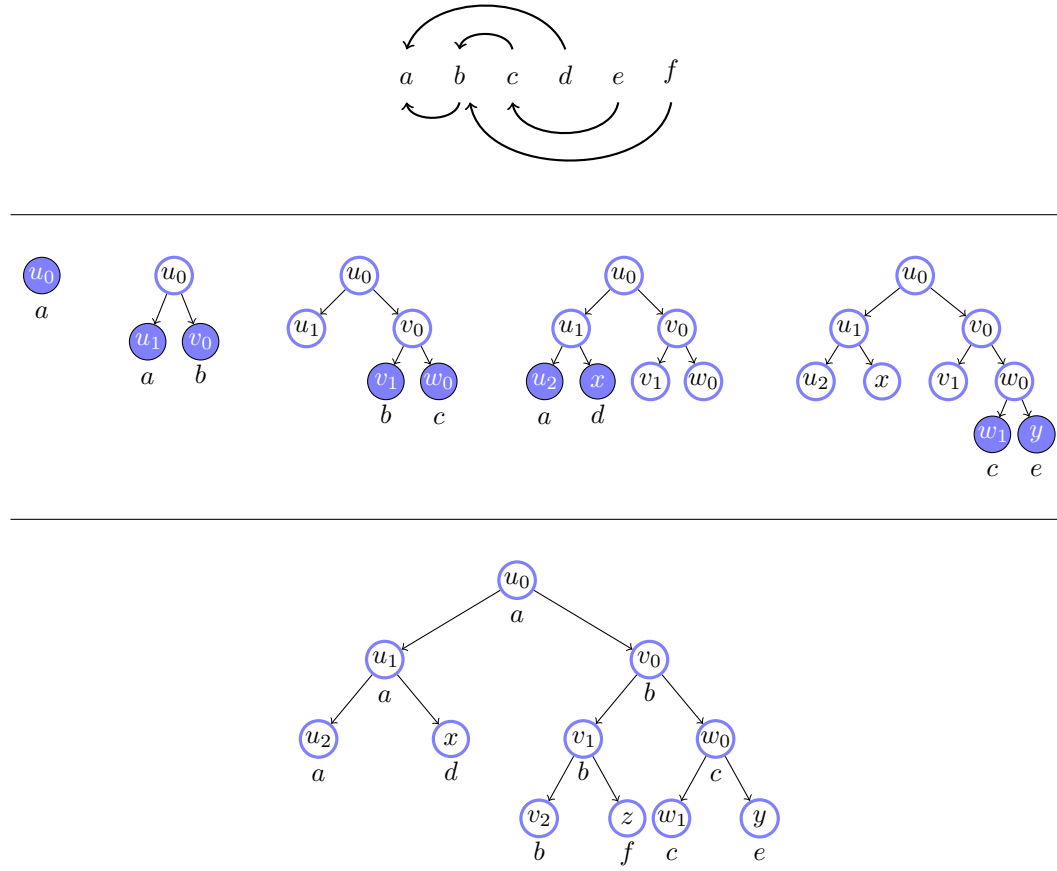
Figure 2 In this figure, a ball tree is computed for a given permutation and predecessor pairing. The permutation (a, b, c, d, e, f) is depicted with arrows representing a predecessor mapping at the top. The tree is constructed incrementally in the middle. Each new point creates two new nodes. The centers of newly inserted nodes are show below the nodes. The completed tree is shown at the bottom. The center of each node is depicted below it. There may be multiple nodes with the same center.

The algorithm presented above has been analyzed in more detail in Chubet et al. [4]. The following theorem appears as Theorem 5.1 in Chubet et al. [4].

▶ **Theorem 2.** *Let G be a greedy tree with $\alpha > 1$. Then the following properties hold:*
1. *The radius of a node x is bounded, $\mathsf{rad}(x) \leq \frac{\varepsilon_p}{\alpha-1}$.*
2. *Let X be a set of pairwise independent nodes from G. The centers of X are $\frac{(\alpha-1)r}{\alpha}$-packed, where r is the minimum radius of any parent of a node in X.*
3. *The height of G is $2^{O(d)} \log \Delta$.*

Theorem 2 allows us to bound the running time of the greedy tree construction algorithm. Constructing the tree topology takes $O(n)$ time and computing radii takes $O(n \log \Delta)$ time. Therefore, given a greedy permutation and the approximate nearest predecessors, the corresponding greedy tree is computed in $O(n \log \Delta)$ time.

Once the tree G is constructed for set A, it is traversed using a max heap that stores the nodes using the radius as the key. The heap H is initialized with the root node. The traversal continues while there is a node in H with non-zero radius. In each iteration, the
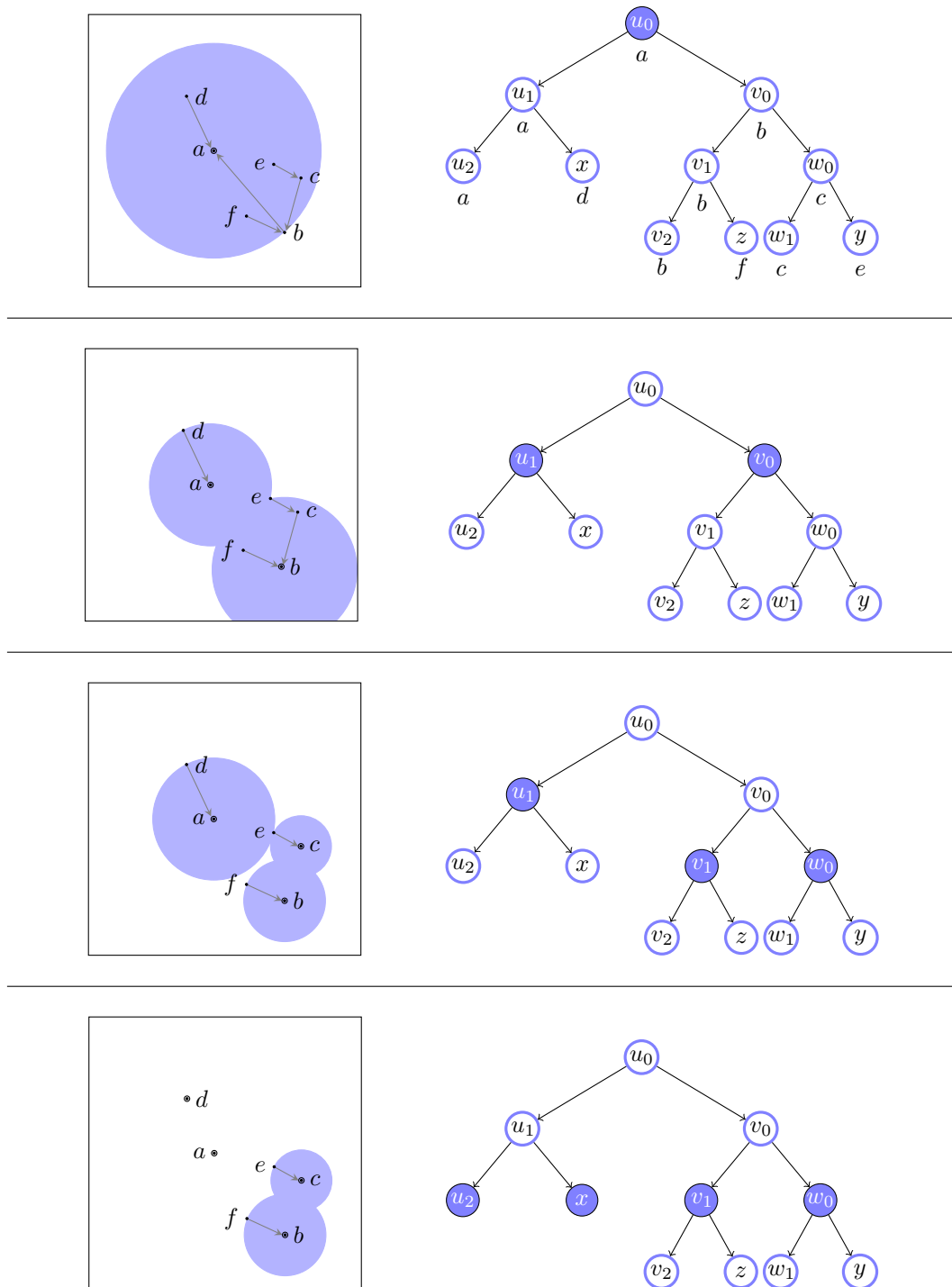
Figure 3 This figure shows a greedy tree and the first four iterations of its radius-order traversal. The center of every node is shown below the node in the first figure. In this tree $\mathsf{ctr}(u_0) = \mathsf{ctr}(u_1) = \mathsf{ctr}(u_2)$ but $\mathsf{rad}(u_0) \neq \mathsf{rad}(u_1) \neq \mathsf{rad}(u_2)$. Also, $\mathsf{pts}(u_0) = \{a, b, c, d, e, f\}$ while $\mathsf{pts}(u_1) = \{a, d\}$ and $\mathsf{pts}(v) = \{b, c, e, f\}$. The heap is initialized with u_0. In each iteration, the node with the largest radius is replaced by its children. The complete order of traversal is u_0, v_0, u_1, v_1, w_0.

node with largest radius is removed from H and its children are added to H. We call this the *radius-order traversal* of G. See Figure 3 for an example of this traversal. The following properties hold at every iteration of the radius-order traversal of G:

1. **Covering**: For every point $a \in A$, there exists a unique node $x \in H$ such that $a \in \mathsf{pts}(x)$.
2. **Packing**: For every node in H, the radius of its parent is at least the radius of the node at the top of H. The nodes in H are pairwise independent. So, by Theorem 2, the nodes in H are packed.

4   The directed Hausdorff distance

In this section, we describe HAUSDORFF, our directed Hausdorff distance approximation algorithm. The main input to HAUSDORFF is a pair of greedy trees G_A and G_B built on sets A and B. A simultaneous radius-order traversal of two greedy trees using a single heap can be done as described in the previous section. The result is similar to a dual-tree traversal [5, 6, 8, 14]. This traversal can be done beforehand and the output be stored in a list. We assume we have access to the roots of the two greedy trees and the remaining nodes are stored in a single list sorted by non-increasing radii. The preprocessing time does not exceed the tree construction time. In Section 6 we give an application where this approach is useful.

The input also includes a parameter $\varepsilon > 0$ that determines the approximation factor. The output is the $(1 + \varepsilon)$-approximate directed Hausdorff distance from A to B.

4.1   The setup

The main data structure used is a bipartite graph N. The two parts of N are $N_A \subseteq G_A$ and $N_B \subseteq G_B$. The set of neighbors of node x in N is denoted as $N(x)$. This graph is called the *viability graph* and it satisfies the following invariants:

- **Covering Invariant**: For every point in A (and respectively, B), there exists a unique node in N_A (respectively, N_B) that contains the point as a leaf in its subtree.
- **Edge Invariant**: For every point $a \in A$, if $\mathbf{d}(a, B) = \mathbf{d}(a, b)$ then there is an edge in N between the nodes containing a and b as leaves.

In addition to the graph, HAUSDORFF stores a *local lower bound* on $\mathbf{d}(a, B)$ for each point $a \in A$ that has been added to N. Let node $x \in N_A$. The local lower bound of point $\mathsf{ctr}(x)$, denoted $\ell(x)$, is defined as,

$$\ell(x) := \max \left\{ \min_{y \in N(x)} \mathbf{d}_{\mathsf{ctr}}(x, y) - \mathsf{rad}(y), 0 \right\}.$$

The largest of these lower bounds is stored as the *global lower bound* L. The global lower bound serves as an estimate of the directed Hausdorff distance.

4.2   The algorithm

Now, we describe HAUSDORFF. Let x_0 and y_0 be the root nodes of G_A and G_B respectively. Initialize N to contain x_0 and y_0 as vertices connected by an edge. Initialize $\ell(x_0)$ and L. The main loop iterates over the input list of greedy tree nodes.

Let z be the next node in the list. If $\mathsf{rad}(z) \leq \frac{\varepsilon L}{2}$, then it is safe to ignore the remaining nodes and terminate the loop. We call this inequality the *stopping condition*. Return L as $(1 + \varepsilon)$-approximate directed Hausdorff distance between A and B. Else, let z_l and z_r be the

left and right children of z respectively. Add z_l and z_r as vertices in N. Connect the children to the neighbors of z. Remove z from N. If $z \in N_A$, let $S := \{z_l, z_r\}$, else let $S := N(z_l)$.

For each $x \in S$ iterate over the neighbors of x. An edge (x, y) can be removed if there exists $y' \in N(x)$ such that

$$\mathbf{d}_{\mathsf{ctr}}(x, y) - \mathsf{rad}(y) > \mathbf{d}_{\mathsf{ctr}}(x, y') + 2 \cdot \mathsf{rad}(x).$$

Remove all such edges incident to x (see Figures 4 and 5). We refer to this step as *pruning* node x. Finally, update $\ell(x)$. A lower bound update is shown in Figure 6. Repeat the main loop until the stopping condition is satisfied.
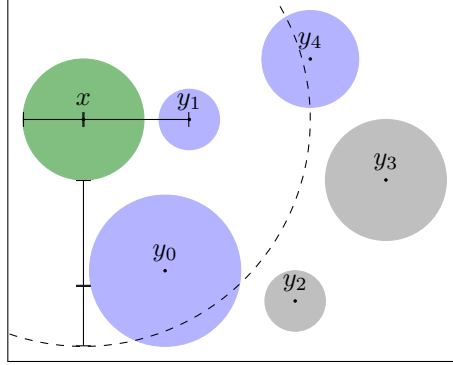
Figure 4 The the nodes y_2 and y_3 are too far away to contain the nearest neighbor of any point in x, so edges (x, y_2) and (x, y_3) can be pruned from the viability graph. The pruning condition respects the neighbor invariant and does not prune edge (x, y_4).

4.3   Analysis

Figure 7 illustrates how the pruning condition does not violate the Edge Invariant. This is formalized in the following lemma:

▶ **Lemma 3.** *For $a \in A$, let $b \in B$ such that $\mathbf{d}(a, B) = \mathbf{d}(a, b)$. Let $x \in N_A$ and $y \in N_B$ be such that $a \in \mathsf{pts}(x)$ and $b \in \mathsf{pts}(y)$ in some iteration of HAUSDORFF. Then, there exists an edge between x and y in N.*

Proof. Given $a \in A$, let $b \in B$ such that $\mathbf{d}(a, b) = \mathbf{d}(a, B)$. By the Covering Invariant, there exists $x \in N_A$ and $y \in N_B$ such that $a \in \mathsf{pts}(x)$ and $b \in \mathsf{pts}(y)$. Then, by the triangle inequality, $\mathbf{d}(a, b) \geq \mathbf{d}_{\mathsf{ctr}}(x, y) - \mathsf{rad}(x) - \mathsf{rad}(y)$. Suppose for the sake of contradiction, the edge (x, y) in N was pruned in the current iteration. Then (x, y) must satisfy the pruning condition, i.e., for some $y' \in N_B$ we have $\mathbf{d}_{\mathsf{ctr}}(x, y) - \mathsf{rad}(y) > \mathbf{d}_{\mathsf{ctr}}(x, y') + 2 \cdot \mathsf{rad}(x)$. It follows that $\mathbf{d}(a, b) > \mathbf{d}_{\mathsf{ctr}}(x, y') + \mathsf{rad}(x)$. Additionally, $\mathbf{d}_{\mathsf{ctr}}(x, y') \geq \mathbf{d}(a, \mathsf{ctr}(y')) - \mathsf{rad}(x)$ by the triangle inequality. Thus, we have $\mathbf{d}(a, b) > \mathbf{d}(a, \mathsf{ctr}(y'))$. This is a contradiction as b is a nearest neighbor of a. Therefore, there exists an edge between x and y. ◄

▶ **Lemma 4.** *Let r be the radius of the node to be processed in an iteration of HAUSDORFF and let L be the global lower bound. Then, $L \leq \mathbf{d}_h(A, B) \leq L + 2r$. Moreover, if $r \leq (\frac{\varepsilon}{2})L$, then L is a $(1 + \varepsilon)$-approximation of $\mathbf{d}_h(A, B)$.*
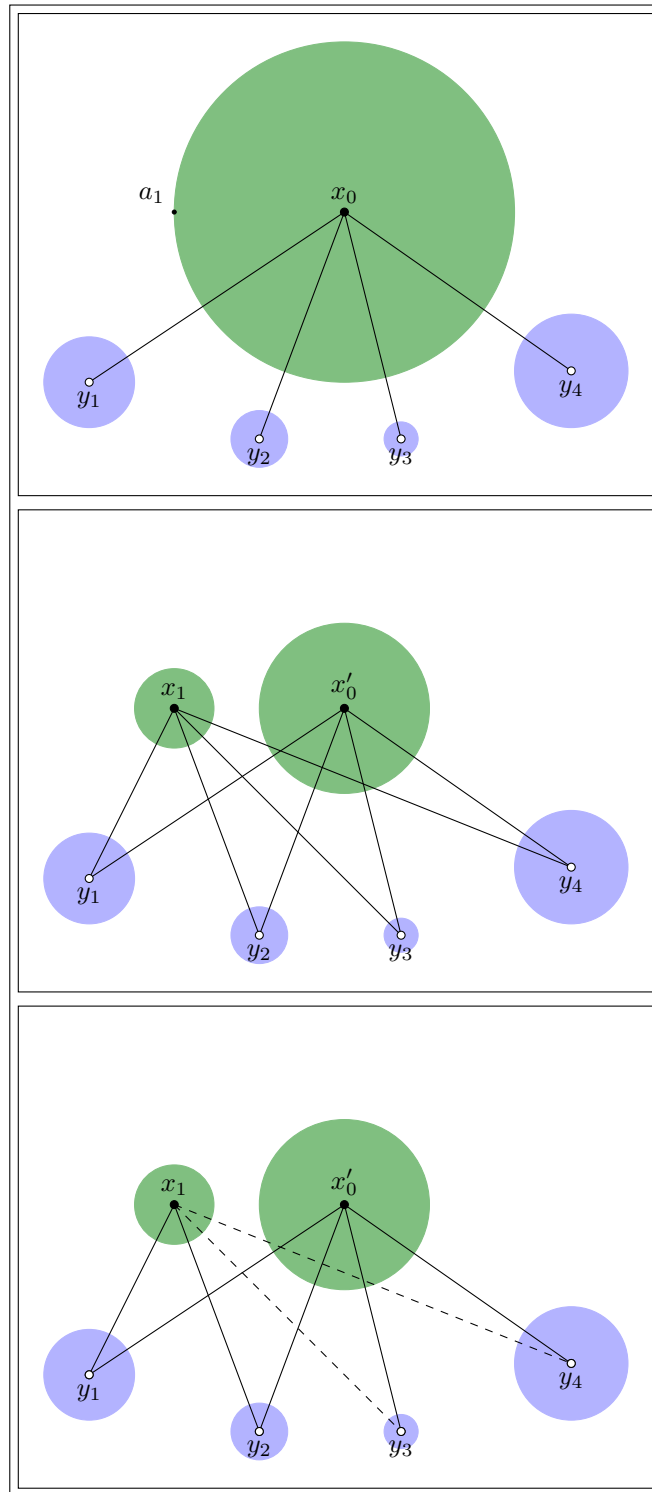
Figure 5 This figure illustrates pruning in an iteration of Hausdorff. At the top, the node x_0 is connected to nodes y_1, y_2, y_3, and y_4. When x_0 is split and replaced by its children (middle image), edges are added between the new node x_1 and the neighbors of its parent. Then, at the bottom, the edges that are too long are pruned using the pruning condition. In this case, (x_1, y_3) and (x_1, y_4) are pruned.
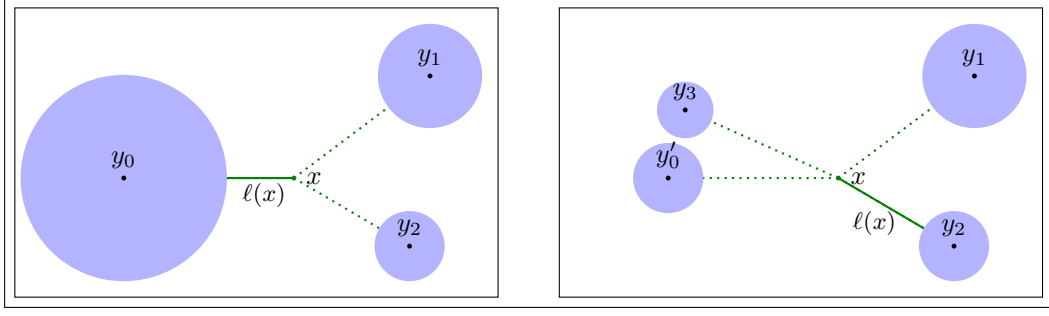
Figure 6 This figure depicts an update of $\ell(x)$ after replacing a node y_0 with its children.
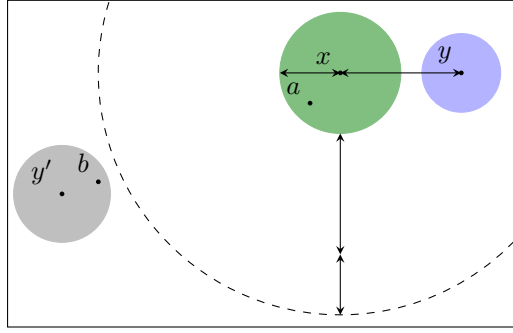
Figure 7 Let $x \in N_A$. For every point a in $\mathsf{pts}(x)$, we have $\mathbf{d}(a, B) \leq \mathbf{d}_{\mathsf{ctr}}(x, y) + \mathsf{rad}(x)$ by the triangle inequality. If edge (x, y') has been pruned, then no point $b \in \mathsf{pts}(y')$ can be the nearest neighbor of all $a \in \mathsf{pts}(x)$, because $\mathbf{d}(a, \mathsf{ctr}(y)) \leq \mathbf{d}(a, b)$ for any such b.

Proof. Let $A' := \{\mathsf{ctr}(x) \mid x \in N_A\}$ and $B' := \{\mathsf{ctr}(y) \mid y \in N_B\}$. We first show that the distance from A' to B is at most $L + r$. We know that $\mathbf{d}_h(A', B) \leq \mathbf{d}_h(A', B')$ because for any $a \in A'$, we have $\mathbf{d}(a, B) \leq \mathbf{d}(a, B')$. Also, $\mathsf{rad}(z) \leq r$ for any vertex $z \in N$. So,

$$\mathbf{d}(a, B') \leq \min_{y \in N_B} \{\mathbf{d}(a, \mathsf{ctr}(y)) - \mathsf{rad}(y) + r\} \leq \ell(x) + r,$$

where $x \in N_A$ is the node with $\mathsf{ctr}(x) = a$. It follows that,

$$\mathbf{d}_h(A', B) \leq \max_{x \in N_A} \{\ell(x) + r\} = L + r.$$

Furthermore, $\mathbf{d}(a, B) \leq \mathbf{d}(a, A') + \mathbf{d}_h(A', B)$, and we know that $\mathbf{d}(a, A') \leq r$. So, $\mathbf{d}(a, B) \leq L + 2r$. Therefore $L \leq \mathbf{d}_h(A, B) \leq L + 2r$. It follows that if $r \leq \frac{\varepsilon}{2}L$ then $\mathbf{d}_h(A, B) \leq (1 + \varepsilon)L$. ◀

At all times, the degree of every vertex in N is bounded by a constant. We will show that this invariant is guaranteed by the pruning algorithm, the early stopping condition, and a packing bound. It is the critical fact in the following analysis of the HAUSDORFF running time.

▶ **Theorem 5.** *Given two greedy trees for sets A and B of total cardinality n, HAUSDORFF computes a $(1 + \varepsilon)$-approximation of $\mathbf{d}_h(A, B)$ in $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)} n$ time.*
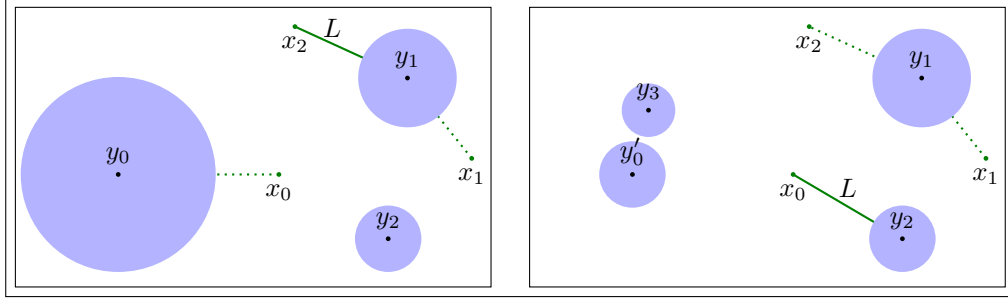
Figure 8 This figure depicts an update of L after replacing the node y_0 with its children. The directed Hausdorff distance, $\mathbf{d}_h(A, B) \geq L$.

Proof. Lemmas 3 and 4 imply the correctness of HAUSDORFF. Now, we bound the running time. Let G_A and G_B be α-approximate greedy trees. In order to bound the degrees of the viability graph N, we first establish that the points associated with the neighbors of a vertex in N are packed. Let r be the radius of the next node in the input list. By construction, any node $z \in N$ is the left or right child of a greedy tree node with radius at least r. Then by Theorem 2, the centers of nodes in N are $\frac{(\alpha-1)r}{\alpha}$-packed. The pruning condition implies that the distance from $\mathsf{ctr}(z)$ to any of its neighbors is at most $L + 4r$. Thus, by Lemma 1,

$$|N(z)| \leq \left(\frac{2\alpha(L + 4r)}{(\alpha - 1)r}\right)^d \leq \left(\frac{16\alpha(1 + \varepsilon)}{(\alpha - 1)\varepsilon}\right)^d,$$

for all $r \geq (\frac{\varepsilon}{2})L$. Therefore, the number of edges incident to any given node in N is $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)}$. So we spend $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)}$ time for each iteration of the algorithm. This gives a running time of $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)} n$. ◄

5 The partial directed Hausdorff distance

Let A and B be subsets of a metric space $(X, \mathbf{d})$. Let $A^{(k)}$ denote all subsets of A with k elements removed. That is, $A^{(k)} := \{S \subseteq A : |A \backslash S| = k\}$. The kth-partial directed Hausdorff distance is $\mathbf{d}_h^{(k)}(A, B) := \min_{S \in A^{(k)}} \mathbf{d}_h(S, B)$. Equivalently, $\mathbf{d}_h^{(k)}(A, B)$ is the $(k+1)^{st}$ largest distance $\mathbf{d}(a, B)$ over all $a \in A$. In particular, $\mathbf{d}_h^{(0)} = \mathbf{d}_h$, as shown in Figure 9.
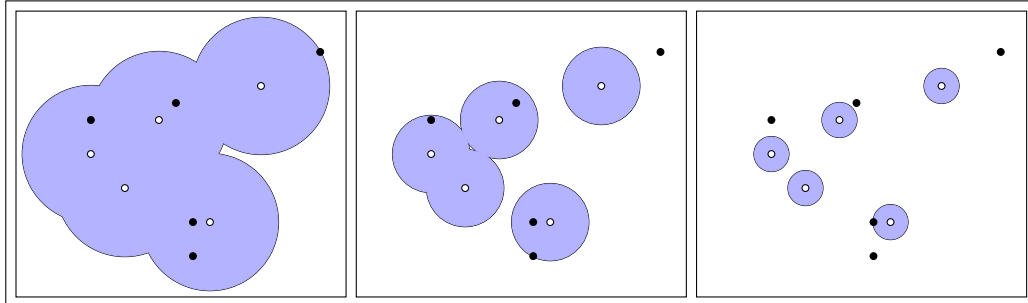
Figure 9 Depicted above are the directed Hausdorff distance $\mathbf{d}_h(A, B)$ (left), the first partial Hausdorff distance $\mathbf{d}_h^{(1)}(A, B)$ (center), and the 4th-partial Hausdorff distance $\mathbf{d}_h^{(4)}(A, B)$ (right).

The stopping condition of HAUSDORFF is satisfied when it has discovered a node x with center a such that $\mathbf{d}(a, B)$ is approximately the largest among all points in A. Instead of terminating the loop at this point, if we continue running the algorithm ignoring a, the algorithm discovers the next point in A approximately farthest to B the next time the stopping condition is satisfied. So a simple modification of HAUSDORFF gives us an algorithm to approximate the kth-partial directed Hausdorff distance.

The algorithm K-HAUSDORFF approximates $\mathbf{d}_h^{(k)}(A, B)$ for all $k \leq |A|$. The input to K-HAUSDORFF consists of two greedy trees and an approximation parameter preprocessed in the same way as HAUSDORFF. The output is a sequence $(\delta_0, \ldots, \delta_{n-1})$ of distances such that $\delta_i \leq \mathbf{d}_h^{(i)}(A, B) \leq (1 + \varepsilon)\delta_i$. The running time of the algorithm depends on a novel heap data structure. We present the algorithm first and then discuss data structure choices before analyzing the running time.

5.1 The setup

K-HAUSDORFF maintains the same bipartite viability graph N and local lower bounds as HAUSDORFF. Instead of storing just a global lower bound L, K-HAUSDORFF stores all the nodes with centers in A in a separate max heap H with the local lower bounds as keys. This heap is called the *lower bound heap*. Every local lower bound update also updates the priority of the node in H. Additionally, there is a list to store the output.

5.2 The algorithm

The initialization of K-HAUSDORFF is the same as that of HAUSDORFF. The output list is initialized to be empty.

The main loop of this algorithm iterates over the input list in a manner similar to that of HAUSDORFF. The only change is to the stopping condition. Instead of checking the stopping condition, check the following *finishing condition* at the beginning of each iteration. Let r be the radius of the current node in the input list. Let x be the node at the top of H. While $r \leq \frac{\varepsilon}{2}\ell(x)$ *finish* node x as follows:

1. Remove x from H.
2. Append $\ell(x)$ to the output list for each $a \in \mathsf{pts}(x)$.
3. Remove all edges incident to x in N.

Instead of terminating the loop when the finishing condition is satisfied, as in HAUSDORFF, proceed and update the vertices in N while pruning nodes to remove long edges. The loop runs over the whole input list.

Lower bound heap

In K-HAUSDORFF, the running time of each iteration is determined by the cost of updates to the lower bound heap. Using a standard heap would require $O(\log n)$ time for basic operations. Allowing for a small approximation, we can put the nodes in buckets, where the mth bucket of this heap contains nodes with radii in the interval $(\beta^m, \beta^{m+1}]$, for $1 < \beta \leq 1+\varepsilon$. There are at most $O(\log_\beta \Delta)$ buckets. Using a standard heap on the buckets would lead to $O(\log \log_\beta \Delta)$ time for basic operations. This can be improved still further using a bucket queue [17], an array of the $O(\log_\beta \Delta)$ buckets. Most operations will take constant time except for remove max, which can require iterating through the buckets. We will show that the buckets are removed in non-increasing order so the iteration visits each bucket at most once, incurring a total cost of $O(\log_\beta \Delta)$.
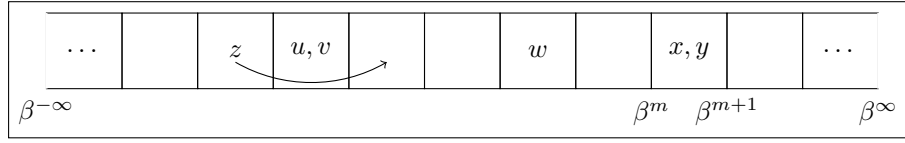
Figure 10 This figure shows how a β-bucket queue implements an approximate lower bound heap as a list of buckets. The number of non-empty buckets is $O(\log \Delta)$. If node x is in the mth bucket then $\beta^m < \ell(x) \leq \beta^{m+1}$. Nodes in the same bucket are accessed in an arbitrary order. Local lower bounds may increase over the course of the algorithm. So a node z may move to a bucket with a higher priority.
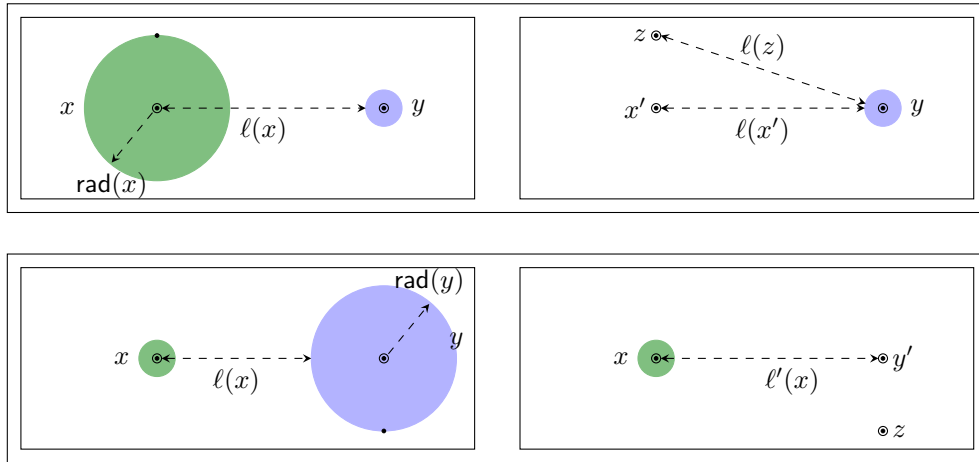
Figure 11 This figure shows the two cases when local lower bounds may increase. At the top, a node x centered at a point in A is replaced by its children, the leaves x' and z. In this case, $\ell(x') = \ell(x)$ and $\ell(z) \leq \ell(x) + \mathsf{rad}(x)$. At the bottom, a node y centered at a point in B is split into its children, the leaves y' and z. Here, we denote the new lower bound of $\mathsf{ctr}(x)$ by $\ell'(x)$. In this case, $\ell'(x) \leq \ell(x) + \mathsf{rad}(y)$.

A problem in using a bucket queue is that local lower bounds may increase over the course of the algorithm. A naïve approach is inefficient as it may end up revisiting buckets. However, the following observation holds by the triangle inequality (see Figure 11).

▶ **Observation 6.** *Let z be the next node to be processed with radius r.*
1. *Let $z \in G_B$. For a neighbor $x \in N(z)$, the local lower bound of x and all descendents of x centered at $\mathsf{ctr}(x)$ will never exceed $\ell(x) + r$ over the course of the algorithm.*
2. *Let $z \in G_A$. Then, the local lower bound of any right child of z will never exceed $\ell(z) + r$ at the end of the iteration. Moreover, the local lower bound of any descendent of z will never exceed $\ell(z) + 2r$ over the course of the algorithm.*

We need to modify the algorithm to avoid revisiting buckets. We can use the following observation.

▶ **Observation 7.** *Let the radius of the next node to be processed be r. Let s be such that $r \leq \frac{\beta-1}{2}\beta^s$. Let x be a node such that $\ell(x) \leq \beta^m$ for some $m \geq s$. Then, the local lower bound of x or any child node of x never exceeds β^{m+1}.*

Proof. It follows from Observation 6 that over the course of the algorithm, the local lower

bound of x never exceeds $\ell(x) + r$. Moreover, for any child of x, the local lower bound never exceeds $\ell(x) + 2r$. By our choice of s and m,

$$\ell(x) + 2r \leq \beta^m + (\beta - 1)\beta^s \leq \beta^{m+1}.$$

Therefore, the local lower bounds never exceed β^{m+1}.   ◀

Now, we describe modified finishing and local lower bound update procedures to use the bucket queue as a lower bound heap. Instead of finishing nodes, we now finish entire buckets. Let $s = \left\lceil \log_\beta\left(\frac{2r\beta}{\beta-1}\right) \right\rceil$. Each time the radius decreases we update s and traverse the array until we reach the new bucket s. For any occupied bucket j encountered before or coinciding with bucket s, append β^j to the output for each $a \in \mathsf{pts}(x)$ for each node x in bucket j. In other words, we finish all nodes with lower bounds greater than β^s. When updating local lower bounds, if there is a node such that its new lower bound would exceed β^s then finish it instead of updating its key.

5.3   Analysis

The analysis of the K-HAUSDORFF algorithm closely follows the analysis of the HAUSDORFF algorithm. As points are removed, the Hausdorff distance decreases, allowing the viability graph to maintain a constant degree throughout as in Theorem 5. The main difference in the running time analysis is that one must include the cost of the heap operations.

First, we show that the output of K-HAUSDORFF is correct.

▶ **Lemma 8.** *Let $(\delta_0, \ldots, \delta_{n-1})$ be the the sequence of distances returned by K-HAUSDORFF. Then, $\delta_i \leq \mathbf{d}_h^{(i)}(A, B) \leq (1 + \varepsilon)\delta_i$ for all $i \in [n]$.*

Proof. Let L be the global lower bound when δ_i is returned. Let $\beta = (1 + \frac{\varepsilon}{2})$. Then, $L \leq \beta\delta_i$. Furthermore, $r \leq \frac{\beta-1}{2}\delta_i$ by the finishing condition. For each j, let p_j denote the jth removed point. Let $S_i = A\backslash\{p_j \mid j \leq i\}$. It follows by the definition of partial Hausdorff distance and Lemma 4 that,

$$\begin{aligned}
\mathbf{d}_h^{(i)}(A, B) &\leq \mathbf{d}_h(S_i, B) \\
&\leq L + 2r \\
&\leq \beta\delta_i + (\beta - 1)\delta_i \\
&= (1 + \varepsilon)\delta_i.
\end{aligned}$$

For the other direction, it is sufficient to show that $\delta_i \leq \mathbf{d}_h^{(i)}(A, B)$. Suppose that $\mathbf{d}_h^{(i)}(A, B) < \delta_i$. Then, there exists some set $S \subset A$ such that $|S| = |S_i|$ and $\mathbf{d}_h(S, A) = \mathbf{d}_h^{(i)}(A, B)$. As $\mathbf{d}_h(S, B) < \delta_i$, for all $a \in S$, we have $\mathbf{d}(a, B) < \delta_i$. By our choice of s and β the δ_i must be non-increasing. Therefore, none of the points in S could have been removed. It follows that $S = S_i$ and this is a contradiction. Therefore, $\delta_i \leq \mathbf{d}_h^{(i)}(A, B) \leq (1 + \varepsilon)\delta_i$ for all $i \in [n]$.   ◀

Now we analyze the running time of K-HAUSDORFF.

▶ **Lemma 9.** *Let $\beta = (1 + \frac{\varepsilon}{2})$. K-HAUSDORFF runs in $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)} n + O(\log_\beta \Delta)$ time.*

Proof. In every iteration, K-HAUSDORFF performs operations on the viability graph and the lower bound heap. It suffices to bound the running time of these operations. The degree of a node in the viability graph is $(2 + \frac{1}{\varepsilon})^{O(d)}$ by the same argument as in Theorem 5.

By Observation 7 and our choice of s, all partial distances can be computed in a single traversal of the lower bound heap. Therefore, traversing the lower bound heap takes $O(\log_\beta \Delta)$ time over the course of the whole algorithm. Thus, the total running time is $(2 + \frac{1}{\varepsilon})^{O(d)} n + O(\log_\beta \Delta)$. ◀

By Lemmas 8 and 9, we conclude the following theorem.

▶ **Theorem 10.** *Let* $\beta = (1 + \frac{\varepsilon}{2})$. K-HAUSDORFF *computes a* $(1 + \varepsilon)$*-approximation of all partial Hausdorff distances in* $\left(2 + \frac{1}{\varepsilon}\right)^{O(d)} n + O(\log_\beta \Delta)$ *time.*

6   An application that amortizes the preprocessing time

The preprocessing time can be amortized in applications where many Hausdorff distance computations are required. For example, in metric multi-dimensional scaling (MDS), all pairwise distances are computed among a sets to give a low-dimensional embedding. HAUSDORFF takes $O(n \log \Delta)$ preprocessing time to construct a greedy tree for each input set. These greedy trees can be re-used in later distance computations. Thus, for large collections of sets, the preprocessing cost does not impact the asymptotic running time. As detailed below, this improves on the running time by a factor of $\log n$ over prior methods.

The goal of MDS is to embed a sample of a metric space into a lower dimensional Euclidean subspace. In this case, the metric space is defined by the Hausdorff distance on subsets of points. Assume we are given k subsets, $X_1, \ldots, X_k$, each of size n. The problem is to compute an MDS embedding of these k sets under the Hausdorff metric. Then to obtain the input for MDS we need to compute the $\binom{k}{2}$ pairwise Hausdorff distances. It takes $O(kn \log \Delta)$ time to compute the k greedy trees, one for each subset X_i. Each distance is computed in $\varepsilon^{-O(d)} n$ time, so the total running time T, including preprocessing time, is

$$T(n, k) = \underbrace{kn \log \Delta}_{\text{greedy tree construction}} + \underbrace{k^2 \varepsilon^{-O(d)} n}_{\text{distance computations}} = kn(\log \Delta + k\varepsilon^{-O(d)}).$$

Thus, for computing the input to MDS, the greedy tree construction time becomes insignificant when $\log \Delta$ is $k\varepsilon^{-O(d)}$. In that case,

$$T(n, k) = k^2 \varepsilon^{-O(d)} n.$$

In comparison with results from the literature, Hausdorff distances can easily be computed in $\varepsilon^{-O(d)} n \log n$ time using any data structure supporting a $\varepsilon^{-O(d)} \log n$ time approximate nearest neighbor search [11, 14, 4, 2]. Thus, one could compute the input for MDS in $O(k^2 n \log n)$ time. Using greedy trees improves this algorithm by a factor of $\log n$.

7   Conclusion

We presented an algorithm to approximate the directed Hausdorff distance that runs in $(2 + \frac{1}{\varepsilon})^{O(d)} n$ time after computing the greedy trees. The benefits of preprocessing outweigh the costs if the same sets are involved in multiple distance computations, such as in MDS computations. With some modifications, the same algorithm can be used to compute all $(1 + \varepsilon)$-approximate k-partial Hausdorff distances in $(2 + \frac{1}{\varepsilon})^{O(d)} n + O(\log_\beta \Delta)$ time.

───── **References** ─────

1 H. Alt, B. Behrends, and J. Blömer. Approximate matching of polygonal shapes. *Annals of Mathematics and Artificial Intelligence*, 13(3-4):251–265, September 1995.

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

3 Y. Chen, F. He, Y. Wu, and N. Hou. A local start search algorithm to compute exact Hausdorff Distance for arbitrary point sets. *Pattern Recognition*, 67:139–148, July 2017.

4 O. Chubet, P. Parikh, D. R. Sheehy, and S. Sheth. Proximity search in the greedy tree. In *2023 Symposium on Simplicity in Algorithms (SOSA)*, pages 332–342.

5 R. Curtin, W. March, P. Ram, D. Anderson, A. Gray, and C. Jr. Tree-independent dual-tree algorithms. *30th International Conference on Machine Learning, ICML 2013*, 04 2013.

6 R. R. Curtin, D. Lee, W. B. March, and P. Ram. Plug-and-play dual-tree algorithm runtime analysis. *Journal of Machine Learning Research*, 16(101):3269–3297, 2015.

7 T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

8 A. Gray and A. Moore. N-Body Problems in Statistical Learning. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000.

9 S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

10 D. Huttenlocher, G. Klanderman, and W. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.

11 R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 798–807, USA, 2004. Society for Industrial and Applied Mathematics.

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

13 S. M. Omohundro. Five balltree construction algorithms. Technical Report 562, ICSI Berkeley, 1989.

14 P. Ram, D. Lee, W. March, and A. Gray. Linear-time algorithms for pairwise statistical problems. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

15 J. Ryu and S.-i. Kamata. An efficient computational algorithm for Hausdorff distance based on points-ruling-out and systematic random sampling. *Pattern Recognition*, 114:107857, June 2021.

16 D. R. Sheehy. greedypermutations. https://github.com/donsheehy/greedypermutation, 2020.

17 S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London Ltd, 2008.

18 A. A. Taha and A. Hanbury. An Efficient Algorithm for Calculating the Exact Hausdorff Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(11):2153–2163, November 2015.

19 D. Zhang, F. He, S. Han, L. Zou, Y. Wu, and Y. Chen. An efficient approach to directly compute the exact Hausdorff distance for 3D point sets. *Integrated Computer-Aided Engineering*, 24(3):261–277, July 2017.