An Incremental Hausdorff Distance Calculation Algorithm

Sarana Nutanong Edwin H. Jacox Hanan Samet

Center for Automation Research, Institute for Advanced Computer Studies
Department of Computer Science, University of Maryland
College Park, Maryland 20742
{nutanong, jacox, hjs}@cs.umd.edu

ABSTRACT

The Hausdorff distance is commonly used as a similarity measure between two point sets. Using this measure, a set X is considered similar to Y iff every point in X is close to at least one point in Y. Formally, the Hausdorff distance $\text{HAUSDIST}(X, Y)$ can be computed as the MAX-MIN distance from X to Y, i.e., find the maximum of the distance from an element in X to its nearest neighbor (NN) in Y. Although this is similar to the closest pair and farthest pair problems, computing the Hausdorff distance is a more challenging problem since its MAX-MIN nature involves both maximization and minimization rather than just one or the other. A traditional approach to computing $\text{HAUSDIST}(X, Y)$ performs a linear scan over X and utilizes an index to help compute the NN in Y for each $\boldsymbol{x}$ in X. We present a pair of basic solutions that avoid scanning X by applying the concept of aggregate NN search to searching for the element in X that yields the Hausdorff distance. In addition, we propose a novel method which incrementally explores the indexes of the two sets X and Y simultaneously. As an example application of our techniques, we use the Hausdorff distance as a measure of similarity between two trajectories (represented as point sets). We also use this example application to compare the performance of our proposed method with the traditional approach and the basic solutions. Experimental results show that our proposed method outperforms all competitors by one order of magnitude in terms of the tree traversal cost and total response time.

1. INTRODUCTION

The Hausdorff distance is a measure of the maximum of the minimum distance between two sets of objects. The problem of determining the MAX-MIN distance over two sets may arise in spatial problems that require a similarity measure between two point sets. For example, the Hausdorff distance can be used as a service coverage measure. In particular, given a set W of warehouses and a set C of customers' locations, the Hausdorff distance from C to W corresponds to the greatest distance that a customer has to travel to their nearest warehouse. As can be seen, this problem involves (i) finding the minimum distance to a warehouse for each customer, and (ii) finding the maximum of the minimum distances. The Hausdorff distance also corresponds to the last pair in the results of the distance semi-join of W and C [16]. That is, $\text{MAX}\{\text{MINDIST}(\boldsymbol{c}, W) : \boldsymbol{c} \in C\}$. The MAX-MIN nature of the Hausdorff distance makes it a more challenging problem than the closest and farthest pair problems which involve minimization or maximization alone, but not both.

The Hausdorff distance is commonly used in similarity determination of two shapes [17] and measuring errors in creating a triangular mesh for approximating a surface [4]. In this paper, our motivating application is trajectory matching. Intuitively, we can consider the Hausdorff distance $\text{HAUSDIST}(A, B)$ as the worst-case discrepancy of one trajectory A with respect to another trajectory B. Using this measure, a trajectory A is considered similar to B *iff* every point in A is close to at least one point in B. To demonstrate the effectiveness of this measure, example search results of trajectories in a real road network are given in Appendix F.

Figure 1 illustrates how $\text{MINDIST}(\boldsymbol{a}, B)$ for each $\boldsymbol{a}$ in A can be evaluated by finding the nearest neighbor (NN) in B. This definition shows that $\text{HAUSDIST}(A, B)$ is asymmetrical. The symmetrical Hausdorff distance $\text{SYMHAUSDIST}(A, B)$ is defined as $\text{MAX}\{\text{HAUSDIST}(A, B), \text{HAUSDIST}(B, A)\}$. An example of $\text{SYMHAUSDIST}(A, B)$ is provided in Appendix A.

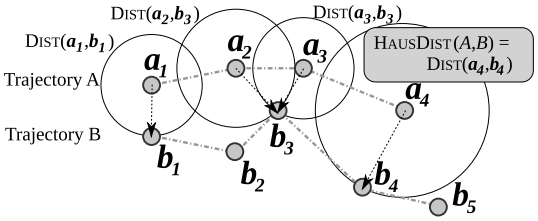


Figure 1: Hausdorff Distance $\text{HAUSDIST}(A, B)$

One example use of the distance measure in this problem domain is bus route comparison. Assume that a transport authority is considering whether to replace one bus route A by another route B. Intuitively, we want the route A to have a low discrepancy with respect to B. Given that the two bus routes are represented as locations of stops, the Hausdorff distance $\text{HAUSDIST}(A, B)$ can be used to indicate the worst-case distance that any customer of route A would need to walk from their usual stop in route A to its nearest stop in route B, should the route replacement take place.

In this paper, we present three algorithms (two basic methods and one proposed algorithm), which utilize hierarchical indexes and the branch-and-bound search principle. The two basic methods are extensions to the concept of *aggregate nearest neighbor querying* [22]. To compute $\text{HAUSDIST}(A, B)$, both of these methods consider the trajectory B as a single object and traverses the index of A to identify the point $\boldsymbol{a}$ in A that yields $\text{MAX}\{\text{MINDIST}(\boldsymbol{a}, B) : \boldsymbol{a} \in A\}$.

Proceedings of the VLDB Endowment, Vol. 4, No. 8

Our proposed method, on the other hand, traverses the indexes of A and B simultaneously to find one point in A and another point in B that yield HAUSDIST(A, B). As mentioned earlier, calculating HAUSDIST(A, B) involves both minimization and maximization. Hence, we cannot apply existing methods [16, 25, 27] which traverse indexes of two point sets to find closest pairs.

The novelty of our method lies in the proposed data structure which is a two-level hierarchy of priority queues where (i) the main level arranges entries in descending order and controls the traversal order of the index of A; (ii) the secondary level arranges entries in ascending order and controls the traversal order of the index of B. This setup corresponds to the MAX-MIN nature of the distance function. The benefit of this approach is demonstrated by our experimental results which show that the proposed method significantly outperforms the basic methods by an order of magnitude.

The contributions of this paper are as follows: (i) Basic branch-and-bound algorithms and a proposed method to compute HAUSDIST between two point sets. (ii) A method to browse trajectories in increasing order of HAUSDIST. (iii) Experimental studies which show that our proposed method outperforms the traditional and basic branch-and-bound algorithms by one order of magnitude.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 contains the equations that are required to estimate the Hausdorff distance for objects in minimum bounding rectangles (MBRs). Section 4 presents basic methods based on the principles of depth-first and best-first traversal. Sections 5 and 6 describe our proposed HAUSDIST algorithm and an algorithm to browse trajectories based on increasing HAUSDIST, respectively. Section 7 contains results of experimental studies. Concluding remarks are drawn in Section 8.

2. RELATED WORK

2.1 Trajectory Comparison

The L_p distance provides a classical means to measure similarity between two trajectories. For example, letting trajectories A and B denote point sequences $\langle \boldsymbol{a_1}, ..., \boldsymbol{a_n} \rangle$ and $\langle \boldsymbol{b_1}, ..., \boldsymbol{b_n} \rangle$, respectively, the L_2 distance between A and B can be derived from the following expression: $\left(\sum_{i=1}^{n} \|\boldsymbol{a_i} - \boldsymbol{b_i}\|^2\right)^{(1/2)}$. By using this measure, we assume that (i) a point $\boldsymbol{a_i}$ in A always corresponds to $\boldsymbol{b_i}$ in B; (ii) A and B contain the same number of points. These assumptions can be relaxed by introducing "elasticity" into the way in which the locations in A are matched against the locations in B. In particular, elasticity is used by allowing each point $\boldsymbol{a_i}$ in A to match $\boldsymbol{b_j}$ in B where the difference between i and j is smaller than a specific temporal threshold [26] or temporally stretching/contracting parts of A to find matching subsequences in B [9, 20].

To mitigate the effects of noise, the concept of edit distance can be applied. For example, given a point $\boldsymbol{a}$ in A and a matching point $\boldsymbol{b}$ in B, one can use a binary weight to indicate if the distance between $\boldsymbol{a}$ and $\boldsymbol{b}$ is small enough to be considered as the same point instead of the actual distance. Notable methods that utilize the edit distance concept include: *edit distance with real penalty* [10], *edit sequence on real sequence* [11], *time warp edit distance* [21], and *longest common subsequence* [26].

We now consider the Hausdorff distance. In terms of elasticity, since each point $\boldsymbol{a}$ in A is allowed to match with any point in B, the default definition of HAUSDIST(A, B) is completely elastic. This means that we are seeking the maximum discrepancy rather than attempting to match each point in A to a point in B as is the case with methods based on the edit distance concept. Note that if less elasticity is desirable, then we can introduce a temporal constraint which allows $\boldsymbol{a}$ to match with only a subset of B within a certain time window. As mentioned earlier, we must contend with the presence of noise which may lead to the existence of outliers. The effects of the noise can be mitigated by ignoring the outliers through the use of partial matching techniques [18] which calculate the m-th partial Hausdorff distance. Specifically, the m-th partial Hausdorff distance (m-HAUSDIST) is the m-th smallest distance from each a point $\boldsymbol{a}$ in A to B.[1] As a result, possible noise is excluded from the resultant m-HAUSDIST by ignoring points $\boldsymbol{a}$ in A that produce the l greatest distances MINDIST$(\boldsymbol{a}, B)$, where l is equal to $(|A| - m - 1)$. The value of l can be determined by the number of outliers. For example, if we know that less than 1% of the measurements in A are affected by noise, then the value of l can be set to $(|A| \times 0.01)$.

2.2 Hausdorff Distance Computation

Alt et al. [3] present a method to compute HAUSDIST from one polygon A to another polygon B using a Voronoi diagram [6]. Their method uses the vertices and edges of B as generators for a Voronoi diagram which is then used to index the edges of B in computing the Hausdorff distance from A to B. This is done by iterating through each of the edges a of A to compute the Hausdorff distance from a to B and retaining the maximum of these distances.

In applications where object locations are not fixed such as image/shape matching, a more robust use of the Hausdorff distance involves the calculation of the minimum Hausdorff distance under rotation and translation [17]. Due to the relatively high computational complexity of this algorithm, approximation methods have also been studied [2, 19].

In this paper, we are interested in calculating the Hausdorff distance between two sets of points with fixed locations (e.g., bus stops of a bus route). Hence, techniques [1, 2, 17, 19] which utilize rotations and translations are irrelevant to our problem. The objective of our research is to apply the branch-and-bound principle to improve on the existing approach [3] to calculate HAUSDIST(A, B) by not having to examine each element $\boldsymbol{a}$ in A.

2.3 Branch-and-Bound Search over two Sets of Points

The aggregate nearest neighbor problem is concerned with finding a point $\boldsymbol{p}$ in a dataset $\mathcal{D}$ which minimizes the aggregate distance with respect to a set Q of query points. That is, finding a point $\boldsymbol{p}$ that yields the minimum value of the set $\{\text{AGG}\{\text{DIST}(\boldsymbol{p}, \boldsymbol{q}) : \boldsymbol{q} \in Q\} : \boldsymbol{p} \in \mathcal{D}\}$, where AGG is an aggregate function: MAX, MIN or SUM. For example, given a set Q of user locations and a set $\mathcal{D}$ of possible meeting locations, the aggregate function MIN can be used to find a meeting location nearest to any of the users in Q. Papadias et al. [22] propose a method which (i) treats Q as one query object, and (ii) finds an object that minimizes the aggregate distance to Q by searching $\mathcal{D}$ in a manner similar to the NN query [16, 23].

The *closest-pair query* in spatial databases [12] involves finding two objects from two different datasets where the distance between them is minimized. Formally, the closest pair of two point sets X and Y is a pair $(\boldsymbol{x}, \boldsymbol{y})$ that yields the minimum value of the set $\{\text{DIST}(\boldsymbol{x}, \boldsymbol{y}) : \boldsymbol{y} \in Y, \boldsymbol{x} \in X\}$.

In a more general setting, the *incremental distance join (IDJ)* and *incremental distance semi-join (IDSJ)* problems [15, 25, 27] are concerned with finding the closest pair and next closest ones in an incremental order of the distance. The difference between IDJ and IDSJ is that when running to completion IDJ iterates through all possible pairs. On the other hand, IDSJ of two sets X and Y produces only pairs $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x} \in X$, $\boldsymbol{y} \in Y$ and $\boldsymbol{y}$ is the nearest neighbor of $\boldsymbol{x}$.

[1] For example, $|A|$-HAUSDIST(A, B) is HAUSDIST(A, B); and 1-HAUSDIST(A, B) is MINDIST(A, B).

According to this definition, the IDSJ and aggregate NN queries (with the aggregate function of MIN) produce comparable results. Specifically, when running to completion, the last pair produced by the IDSJ of X and Y is the pair that yields the distance

$$\text{MAX}\{\text{MINDIST}(\boldsymbol{x}, Y) : \boldsymbol{x} \in X\}. \quad (1)$$

Similarly, the object in $\mathcal{D}$ which maximizes the MIN-aggregate distance with respect to the query set Q has the distance of

$$\text{MAX}\{\text{MINDIST}(\boldsymbol{p}, Q) : \boldsymbol{p} \in \mathcal{D}\}. \quad (2)$$

Although the two expressions (1) and (2) are equivalent, these two problems are different in practice. The coverage and cardinality of the aggregate NN query set Q are usually smaller than those of the dataset $\mathcal{D}$, while the two datasets X and Y in an IDSJ query usually have similar coverage and cardinality. As a result, an aggregate NN query is often processed by representing Q as a single query object and performing a traversal on the index of $\mathcal{D}$. On the other hand, processing a IDSJ query requires indexes for both X and Y, and involves traversing the two indexes simultaneously.

The two expressions (1) and (2) are also equivalent to the HAUSDIST definition given in Section 1. According to this observation, the HAUSDIST calculation can be done by applying the principle of *farthest point* querying [24] so that the object with the greatest MINDIST is returned first. Basic branch-and-bound search methods based on this principle are presented in Section 4.

3. HAUSDORFF DISTANCE ESTIMATORS

In order to apply the branch-and-bound concept to the computation of the Hausdorff distance, we derive upper and lower bounds of HAUSDIST in this section. We assume an R-tree like index structure [7, 8, 13], although a similar analysis could be applied to any hierarchical containment index for spatial data [5]. A lower bound of the Hausdorff distance from one MBR A to another MBR B can be determined by exploiting the fact that for each MBR face, there has to be at least one object that touches it. By this means, we can compute the smallest possible NN distance that each MBR face can produce and the overall maximum of these estimates is guaranteed to be lower than the actual Hausdorff distance. We formally define a lower bound of the Hausdorff distance as follows.

DEFINITION 1 (LOWER BOUND). *Given two MBRs A and B, a lower bound of the Hausdorff distance from the elements confined by A to the elements confined by B is defined as*

$$\text{HAUSDISTLB}(A, B) = \text{MAX}\{\text{MINDIST}(f_a, B) : f_a \in \text{FACESOF}(A)\}.$$

As shown in Figure 2, $\text{HAUSDISTLB}(A, B)$ is calculated by evaluating the MINDIST from each face in A to the MBR B. The maximum, i.e., MINDIST(A.LEFT, B), of these values is the lower bound of the Hausdorff distance.

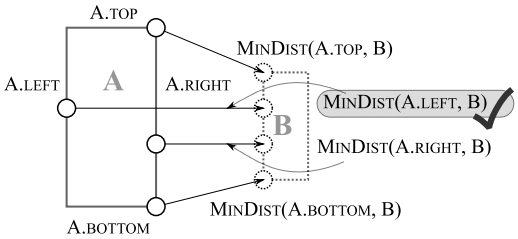


Figure 2: Lower bound of HAUSDIST from A to B.

Similarly, an upper bound can be computed as the greatest possible NN distance that each MBR face could produce and the overall maximum of these estimates is guaranteed to be greater than the actual Hausdorff distance.

DEFINITION 2 (UPPER BOUND). *Given two MBRs A and B, an upper bound of the Hausdorff distance from the elements confined by A to the elements confined by B is defined as*

$$\text{HAUSDISTUB}(A, B) = \text{MAX}\{\text{MAXNEARESTDIST}(f_a, B) : f_a \in \text{FACESOF}(A)\},$$

where the MAXNEARESTDIST *from an MBR face f_a to an MBR B is given as* $\text{MIN}\{\text{MAXDIST}(f_a, f_b) : f_b \in \text{FACESOF}(B)\}$.

As shown in Figure 3, $\text{HAUSDISTUB}(A, B)$ is calculated by evaluating the MAXNEARESTDIST [24] from each face in A to the MBR B. The maximum MAXNEARESTDIST is the upper bound of the Hausdorff distance.

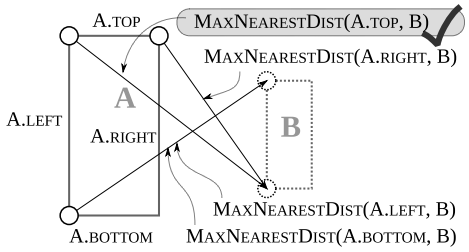


Figure 3: Upper bound of HAUSDIST from A to B.

In the next two sections, we show how these lower and upper bounds are used in branch-and-bound search algorithms to compute the Hausdorff distance from one trajectory to another.

4. BASIC METHODS

Our first basic method is based upon the existing method [3] which scans all elements in X to calculate $\text{HAUSDIST}(X, Y)$. Algorithm 1 visits each point $\boldsymbol{x}$ in X to compute $\text{MINDIST}(\boldsymbol{x}, Y)$ in order to compute $\text{MAX}\{\text{MINDIST}(\boldsymbol{x}, Y) : \boldsymbol{x} \in X\}$. The function DistanceToNN($\boldsymbol{x}$,R_Y) (Line 4) computes $\text{MINDIST}(\boldsymbol{x}, Y)$ using the best-first NN algorithm over R_Y.

Algorithm 1: Scan-HAUSDIST(X, Y)

input : Point Set X, Point Set Y
output: Hausdorff distance from X to Y

```
1 RTree R_Y ← Create an R-Tree for Y;
2 Distance d_H ← 0;
3 for each Point x in X do
4     d_H ← MAX{d_H, DistanceToNN(x, R_Y)} ;
5 return d_H;
```

As can be seen, Algorithm 1 still requires visiting all points in X to calculate $\text{HAUSDIST}(X, Y)$. To mitigate this problem, the branch-and-bound search principle has to be applied to the set X as well. We apply the aggregate NN query technique [22] which considers one of the sets (which is Y in this case) as one single object. Assuming that X is indexed in an R-Tree R_X, depth-first and best-first search on X is done by traversing R_X with respect to lower and upper bounds of the distance to Y. Next, we present two basic branch-and-bound methods to compute $\text{HAUSDIST}(X, Y)$ which traverse the R-Tree R_X of X in depth-first (Algorithm 2) and best-first (Algorithm 3) manners.

The depth-first algorithm is based on the fact that (i) $\text{HAUSDIST}(X, Y)$ can be decomposed into HAUSDIST calculations from subsets of X to Y, and that (ii) the HAUSDIST from a singleton set $\{\boldsymbol{x}\}$ to Y is $\text{MINDIST}(\boldsymbol{x}, Y)$ — the distance from

$\boldsymbol{x}$ to the NN in Y. As shown in Algorithm 2, the algorithm traverses the tree R_X of X in a depth-first manner controlled by recursive function calls. Lines 5 to 22 show a recursive function DF-HAUSDIST* called by Algorithm 2. The recursive function accepts three arguments: a node N_X, an R-Tree R_Y, and a distance lower bound *MaxLB* computed so far. There are two cases depending on whether or not N_X is an object. If N_X is not an object (Lines 8 to 20), we process the children of N_X. This is done by computing the upper bound HAUSDISTUB from each child node to the root node of R_Y (Line 11) and visit them in descending order. As the function traverses nodes in R_X, it keeps track of the maximum result and the maximum lower bound computed from nodes visited so far. The maximum of these two values is stored in the variable *MaxLB* (Line 8 and Line 17). Nodes with upper bounds smaller than *MaxLB* are pruned since they cannot produce a result greater than the current lower bound. Otherwise, if N_X is a point object (Lines 21 to 22), then the distance from N_X to its NN is returned as output. The recursion terminates when all nodes in R_X are visited or pruned.

Algorithm 2: DF-HAUSDIST(X, Y)

```
   input  : Point Set X, Point Set Y
   output : Hausdorff distance from X to Y
1  RTree R_X ← Create an R-Tree for X;
2  RTree R_Y ← Create an R-Tree for Y;
3  MaxLB ← HAUSDISTLB(RootOf(R_X), RootOf(R_Y));
4  return DF-HAUSDIST*(RootOf(R_X), R_Y, MaxLB);

5  Recursive Function DF-HAUSDIST*( N_X, R_Y, MaxLB)
       input  : Node N_X (from R_X), R-Tree R_Y, Maximum
                distance lower bound MaxLB obtained so far
       output : HAUSDIST from objects in N_X to objects in R_Y
6      Distance d_H ← 0;
7      if N_X is a non-object then
8          MaxLB←MAX{MaxLB,HAUSDISTLB(N_X, RootOf(R_Y))};
9          List L ← create an empty list;
10         for each Child Node C of N_X do
11             Distance UB ← HAUSDISTUB(C, RootOf(R_Y));
12             Insert((C, UB), L);
13         Sort L in descending order using the second element;
14         for each (C, UB) in L do
15             if UB ≥ MaxLB then
16                 d_H ←MAX{d_H,DF-HAUSDIST*(C,R_Y,MaxLB)};
17                 MaxLB ← MAX{MaxLB, d_H};
18             else
19                 return d_H;
20         return d_H;
21     else
22         return DistanceToNN(N_X, R_Y);
```

We now present Algorithm 3, a method which traverses the R-Tree for X in best-first order. To calculate the Hausdorff distance HAUSDIST(X, Y) from a point set X to a point set Y, the initialization involves the following steps (Lines 1 to 4):

- create R-Trees R_X and R_Y for X and Y, respectively;
- create an empty priority queue *PQ* which arranges entries (Node N, Distance d_N) in decreasing order according to the key value d_N, where d_N is calculated as (i) an upper bound HAUSDISTUB if N is a non-object, or (ii) the distance from N to its NN in R_Y, otherwise;
- insert the pair (RootOf(R_X), ∞) into *PQ*.

In the while loop (Lines 6 to 16), the best-first search order is controlled by *PQ* which organizes objects according to the key in descending order. At the beginning of each iteration, the entry (Node N, Distance d_N) with the greatest upper bound is retrieved from *PQ*. For each child C of N, we compute the key value d_C and insert (C, d_C) into *PQ*. The while loop iterates until it encounters the first data object $\boldsymbol{p}$. Since the best-first search order guarantees that no other object in PQ can have a greater distance to its NN, the key value of $\boldsymbol{p}$ can be returned as the resultant distance.

Algorithm 3: BF-HAUSDIST(X, Y)

```
   input  : Point Set X, Point Set Y
   output : Hausdorff distance from X to Y
1  RTree R_X ← Create an R-Tree for X;
2  RTree R_Y ← Create an R-Tree for Y;
3  PriorityQueue PQ ← Create a "descending order" priority queue;
4  Insert((RootOf(R_X), ∞), PQ);
5  while PQ is not empty do
6      Entry (N, d_N) ← Dequeue(PQ);
7      if N is a non-object then
8          for each Child Node C of N do
9              Distance d_C;
10             if C is a non-object then
11                 d_C ← HAUSDISTUB(C, RootOf(R_Y));
12             else
13                 d_C ← DistanceToNN(C,R_Y);
14             Insert((C, d_C), PQ);
15     else
16         return d_N;
```

In this section, we have shown that we can apply the concept of aggregate NN query to enable branch-and-bound search over X when calculating HAUSDIST(X, Y). By this means, both depth-first (DF) and best-first (BF) algorithms (Algorithms 2 and 3) avoid examining every single point in the set X. However, the main drawback of these algorithms is that the upper bound calculated on the basis of the root of R_Y may not be an effective estimator (especially when R_X and R_Y are highly overlapped). A more accurate estimator can be calculated from nodes at deeper levels in R_Y, which is more expensive to compute. In the next section, we describe our proposed method which computes a tighter upper bound by exploring R_Y in order of increasing depth.

5. PROPOSED METHOD

Our next objective is to mitigate the upper bound inaccuracy drawback described in the previous section. In this section, we present an algorithm which incrementally explores the indexes of X and Y to calculate the HAUSDIST(X, Y). Similar to the two basic methods, both sets X and Y are assumed to be indexed by R-trees R_X and R_Y, respectively. A priority queue is used to control the order in which nodes N_X in R_X are visited. However, instead of using an upper bound of HAUSDISTUB$(N_X, \text{RootOf}(R_Y))$, a better upper bound is computed from a subset S_N of nodes N_Y inside R_Y. Specifically, an upper bound is computed as

$$\text{MIN}\{\text{HAUSDISTUB}(N_X, N_Y) : N_Y \in S_N\},$$

where S_N is determined by a process which incrementally explores R_Y with respect to the distance from N_X (more details in Section 5.2). The correctness of this new upper bound is verified by Lemma 1 and its associated proof in Appendix C.

The incremental HAUSDIST algorithm (Algorithm 4) uses a while loop to explore both R-Trees R_X and R_Y in a best first manner. In each iteration, a decision is made whether to traverse R_X or R_Y. The order in which nodes in R_X and R_Y are explored is maintained by the data structure illustrated in Figure 4.

5.1 Data Structures

We make use of a hierarchy of priority queues (Figure 4) consisting of one main priority queue (*MainPQ*) and multiple secondary priority queues (*SecPQs*). Specifically, each entry in MainPQ is associated with a SecPQ. MainPQ controls the order in which nodes in R_X are explored. For each MainPQ entry, its corresponding SecPQ controls the order in which nodes in R_Y are explored. Entries in MainPQ are arranged in descending order, i.e., the head entry has the greatest key. Entries in a SecPQ are arranged in ascending order, i.e., the head entry has the smallest key. This setup corresponds to the MAX-MIN nature of the distance function.

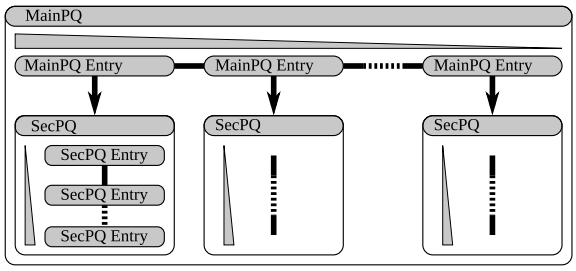


Figure 4: Illustration of the priority queue hierarchy

In the context of HAUSDIST(X, Y) calculation, definitions for priority queue entries of MainPQ and SecPQ are given as follows.

DEFINITION 3 (*MainPQ-Entry*). *Attributes of each MainPQ entry (N_X, UB, SPQ) are described as follows:*

(i) Node N_X *— a node in an R-Tree for X, which could be an object or a non-object (index node);*

(ii) Distance *UB — an upper bound of* HAUSDIST *from data points in N_X to the set Y;*

(iii) SecPQ *SPQ — a priority queue of entries corresponding to nodes of the R-Tree for Y.*

DEFINITION 4 (*SecPQ-Entry*). *Attributes of each SecPQ entry (Node N_Y, Distance LB) are described as follows:*

(i) Node N_Y *— a node in an R-Tree for Y, which could be an index node or a data point;*

(ii) Distance *LB — a lower bound of* HAUSDISTLB(N_X, N_Y)*, where N_X is the node of the MainPQ entry to which the SecPQ entry is associated.*

Note that the upper bound *UB* is calculated as the minimum of HAUSDISTUB(N_X, N_Y) for nodes N_Y in SPQ.

5.2 Algorithm Description

The main algorithm is given by Algorithm 4 and has the following structure. The initialization (Lines 1 to 6) consists of the following steps: (i) create R-Trees R_X and R_Y for point sets X and Y, respectively; (ii) initialize MainPQ *MPQ*; (iii) create a secondary priority queue *SPQ*; (iv) insert the root of R_Y with an initial distance of 0 into *SPQ*; (v) insert the root of R_X with an initial distance of ∞ and a secondary priority queue *SPQ* into *MPQ*. After the initialization, *MPQ* contains a single entry with the root of R_X as the associated node. For each iteration of the while loop (Lines 7 to 15), the head entry (N_X, *UB*, *SPQ*) is dequeued from *MPQ*. We also check the head entry (N_Y, *LB*) of *SPQ* (without dequeuing). Based on N_X and N_Y, a decision is made whether to terminate, to traverse R_X (Algorithm 5) or to traverse R_Y (Algorithm 6).

The case that both N_X and N_Y are data points implies that (i) N_Y is the NN of N_X, since *SPQ* uses HAUSDISTLB as an estimator and arranges its entries in ascending order; (ii) No other entry in *MPQ* can produce an NN distance greater than *UB*, since *MPQ* uses HAUSDISTUB as an estimator and arranges its entries in descending order. As a result, the key value *UB*, which is DIST(N_X, N_Y), can be safely returned as the HAUSDIST from X to Y without having to examine the remaining entries.

Algorithm 4: Inc-HAUSDIST(X, Y)

input : Point Set X, Point Set Y
output: Hausdorff distance from X to Y

```
1  RTree R_X ← Create an R-Tree for X;
2  RTree R_Y ← Create an R-Tree for Y;
3  MainPQ MPQ ← Create a "descending order" PQ;
4  SecPQ SPQ ← Create an "ascending order" PQ;
5  Insert((RootOf(R_Y), 0), SPQ);
6  Insert((RootOf(R_X), ∞, SPQ), MPQ);
7  while MainPQ is not empty do
8      MainPQ-Entry (N_X, UB, SPQ) ← Dequeue(MPQ);
9      SecPQ-Entry (N_Y, LB) ← Head of SPQ;
10     if N_X and N_Y are both points then
11         return UB;
12     else if N_X is farther from the leaf level than N_Y then
13         TraverseX(N_X, SPQ, MPQ)
14     else
15         TraverseY(N_X, UB, SPQ, MPQ)
```

In case that further examination is required, we compare the numbers of hops from the leaf level of N_X and N_Y. Since the goal is to reach two objects, the node farther from the leaf level is chosen for expansion. Specifically, if N_X is farther from the leaf level than N_Y, we deepen the search in R_X by calling *TraverseX* (Algorithm 5). Otherwise, *TraverseY* (Algorithm 6) is called. The description of the deepening process is given as follows.

Algorithm 5: TraverseX(N_X, *SPQ*, *MPQ*)

input : Node N_X in R-Tree of X, SecPQ *SPQ*, MainPQ *MPQ*
output: Modified *MPQ*

```
1  for each Child C of N_X do
2      SecPQ Child-SPQ ← Create an "ascending order" PQ;
3      Distance MinUB ← Initialize to ∞;
4      for each SecPQ-Entry (N_Y, U_LB) in SPQ do
5          Distance LB ← HAUSDISTLB(C, N_Y);
6          Insert((C, LB), Child-SPQ);
7          Distance UB ← HAUSDISTUB(C, N_Y) ;
8          MinUB ← MIN{MinUB, UB}
9      Insert((C, MinUB, Child-SPQ), MPQ);
```

Algorithm 6: TraverseY(N_X, *UB*, *SPQ*, *MPQ*)

input : Node N_X in R-Tree of X, Distance *UB*, SecPQ *SPQ*, MainPQ *MPQ*
output: Modified *MPQ*

```
1  SecPQ-Entry (N_Y, LB) ← Dequeue(SPQ);
2  Distance MinUB ← UB;
3  for each Child Node C of N_Y do
4      Distance LB ← HAUSDISTLB(N_X, C) ;
5      Insert((C, LB), SPQ);
6      Distance UB ← HAUSDISTUB(N_X, C) ;
7      MinUB ← MIN{MinUB, UB}
8  Insert((N_X, MinUB, SPQ), MPQ);
```

Algorithm 5, TraverseX. The algorithm accepts a node N_X, a SecPQ *SPQ* and MainPQ *MPQ*. For each child node C of N_X, a new MainPQ entry is created by the following steps: (i) a new SecPQ *Child-SPQ* is filled with entries in *SPQ* but with lower bounds calculated with respect to C; (ii) the minimum upper bound *MinUB* is calculated as the minimum of HAUSDISTUB(C, N_Y); (iii) a new entry (C, *MinUB*, *Child-SPQ*) is inserted into *MPQ*.

Upon completion of the for loop, each child node C of N_X has a corresponding entry in *MPQ*.

Algorithm 6, TraverseY. The algorithm accepts a node N_X, the current upper bound *UB* of the node N_X, a SecPQ *SPQ* and MainPQ *MPQ*. The first step is to remove the head entry (N_Y, *LB*) from *SPQ*. The next step is to initialize the minimum upper bound *MinUB* to the current upper bound value *UB*. For each child C of N_Y, (i) the lower bound *LB* is calculated as HAUSDISTLB(N_X,C); (ii) a SecPQ entry (C, *LB*) is inserted into *SPQ*; (iii) the upper bound *UB* is calculated as HAUSDISTUB(N_X,C); (iv) the new upper bound is incorporated into the minimum upper bound *MinUB*. A new MainPQ entry (C, *MinUB*, *Child-SPQ*) is inserted into *MPQ*. Output is given via the modification of *MPQ*.

The end product of Algorithm 4 is the directed Hausdorff distance from X to Y. An algorithm to compute SYMHAUSDIST(X, Y) is described in Appendix D.

To mitigate the effect of outliers, one may wish to ignore the first l distances retrieved from MainPQ and return a partial Hausdorff distance [18] instead. This feature can be incorporated into Algorithm 4 by modifying the termination condition (Line 10) so that it returns the $(l + 1)$-th distance instead of the first one.

6. SEARCHING TRAJECTORIES

As stated in Section 1, we want to use the Hausdorff distance as a similarity measure between two trajectories. In this section, we formulate an incremental algorithm for searching similar trajectories based on the Hausdorff distance. We define an incremental search algorithm TRAJSEARCH($Q, \mathcal{D}, T$) (given by Algorithm 8 in Appendix E) as a function that accepts three arguments: a query trajectory Q, a collection $\mathcal{D}$ of trajectories represented by an R-tree index here although other spatial indexes could also be used, and the type T of the Hausdorff distance function. The search algorithm makes use of the R-tree hierarchical index that stores the trajectory set $\mathcal{D}$, and a Hausdorff distance estimator (described in Algorithm 9, Appendix E) to provide the order in which the index is traversed. In this way, we avoid examining all trajectories in the dataset as shown in Appendix G.

Since the distance function can be asymmetrical or symmetrical, our search function supports three search modes, which correspond to the following Hausdorff distance minimization problems.

- FROM Q — MIN$\{$HAUSDIST$(Q, Y) : Y \in \mathcal{D}\}$. For example, given an incomplete trajectory Q of an active cyclone, a meteorologist may wish to find similar cyclone trajectories from a set $\mathcal{D}$ of historic cyclone trajectories. Assume that the results are used for trajectory prediction purposes. The FROM-Q mode can be used to match a shorter trajectory Q with sub-trajectories of longer ones in $\mathcal{D}$.
- TO Q — MIN$\{$HAUSDIST$(Y, Q) : Y \in \mathcal{D}\}$. For example, given a trajectory Q of a bus line, a transportation authority may wish to find k bus lines from a set $\mathcal{D}$ that can be replaced by Q. In this application, we are interested in finding k trajectories that match with a part of Q.
- SYM — MIN$\{$SYMHAUSDIST$(Q, Y) : Y \in \mathcal{D}\}$. The SYM mode can be used for full-trajectory matching. For example, given a set of migration trajectories of different types of birds, zoologists are interested in finding a type of bird that has the same migration pattern as a given type.

To compute the HAUSDIST from one trajectory to another, we can use any of the four algorithms: *Scan*, *Depth-First*, *Best-First* and *Incremental* algorithms given by Algorithms 1, 2, 3 and 4, respectively. Results of an experimental study of these algorithms are given in Section 7.

7. EXPERIMENTAL STUDIES

In this section, we compare the performance of the three basic approaches: *SCAN* (Algorithm 1), *DF* (Algorithm 2) and *BF* (Algorithm 3) with our proposed algorithm, *INC* (Algorithm 4). We chose the R-tree technique [13] for the hierarchical containment index. We used only internal memory and a low fan-out for performance. All algorithms were implemented in C++ and the experiments were performed on a computer with a Core 2 Duo processor and 2GB of Main Memory, running Mac OS 10.5.

We used synthetic and real datasets in our performance studies.

- The synthetic dataset contains trajectories generated from a road network representing main roads in the city of Oldenburg, Germany.
- We used the *GeoLife GPS Trajectories* [28, 29] as our real dataset. The dataset contains trajectories collected from 165 users in a period of 29 months. Each trajectory contains a sequence of locations within a period of one day (12:00:01 am to 11:59:59 pm) of a user. Since we are interested in large trajectories, only trajectories with 3000 locations or more were used in the experiments. This results in 2669 trajectories being used for this dataset.

Performance comparison of the four algorithms was conducted in two different settings: *Hausdorff distance calculation* (Section 7.1) and *Incremental trajectory search* (Section 7.2). The following four cost measures were used.

- *Tree traversal cost* — the number of R-Tree nodes accessed.
- *Distance calculation cost* — the number of MINDIST and MAXDIST calculations. This is because the computation of HAUSDIST lower and upper bounds can be decomposed into MINDIST and MAXDIST calculations. Please refer to Appendix B for the costs for computing a HAUSDIST lower bound and upper bound using MBRs.
- *Priority queue cost* — the total number of comparisons incurred from enqueue and dequeue operations.
- *Total execution time* — the amount of time taken to compute HAUSDIST (Section 7.1) or to find the k most similar trajectories (Section 7.2).

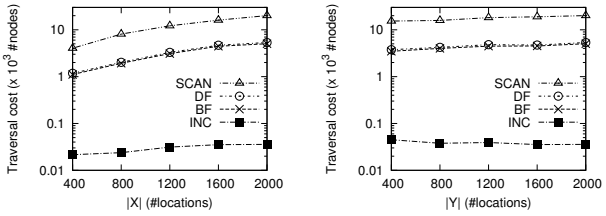
7.1 Hausdorff Distance Calculation

In this set of experiments, we compared the four algorithms to calculate HAUSDIST(X, Y) by varying sizes of X and Y. In order to properly control these two parameters, only the synthetic dataset was used in this experiment. Specifically, we generated trajectories from the Oldenburg road network. Each trajectory was randomly generated as a shortest path with a length of 2000 units (in a data space of 10^4 by 10^4 unit squares). Each trajectory was created in 5 resolutions: 400, 800, 1200, 1600 and 2000 sampled locations. The dataset was organized into 5 sets according to these resolutions. These 5 different sets of trajectories enabled us to vary the parameter $|X|$ on a fixed value of $|Y|$ and vice versa. Each result was recorded as the average of 200 different pairs of trajectories.

Tree traversal cost. Figure 5(a) compares the tree traversal costs of the four algorithms to calculate HAUSDIST(X, Y) as the size $|X|$ changes. Both point sets X and Y are indexed in two R-Trees R_X and R_Y respectively. For all methods, we can see the positive correlation between the traversal cost and $|X|$. This is because an increase in $|X|$ means that there are more NN queries to execute for the three competitors (SCAN, DF and BF) and more MainPQ entries for the proposed method (INC).

Both Figures 5(a) and 5(b) show that INC significantly outperforms all three competitors. The traversal costs of DF and BF are approximately three times lower than that of SCAN. This result shows the effectiveness of a simple improvement which utilizes

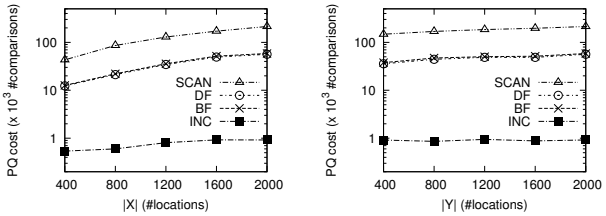
the upper bound computed with respect to the root node of R_Y. Further performance improvement of over an order of magnitude (in comparison to DF and BF) was obtained by INC. This result confirms our hypothesis that a more effective upper bound for HAUSDIST(X, Y) can be obtained by examining deeper nodes in R_Y instead of just the root node.



(a) Effect of $|X|$ (b) Effect of $|Y|$

Figure 5: Traversal costs of calculating HAUSDIST(X, Y)

Priority queue cost. Figure 6 compares the four methods in terms of priority queue cost. The results conform with those of the traversal cost. DF and BF incur similar costs and perform better than SCAN by approximately three times. INC significantly outperforms all competitors.

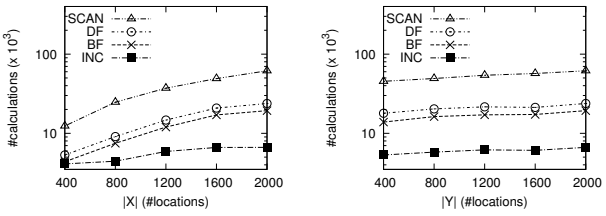


(a) Effect of $|X|$ (b) Effect of $|Y|$

Figure 6: Priority queue costs of calculating HAUSDIST(X, Y)

Distance calculation cost. Figure 7 shows the distance calculation costs of the four methods. With this cost measure, we can see that DF has a slightly higher cost than BF. This is because, DF computes lower bounds of nodes for pruning purposes (Line 8, Algorithm 2), while this step is not required for BF, as pruning is handled implicitly by node scheduling using a priority queue.

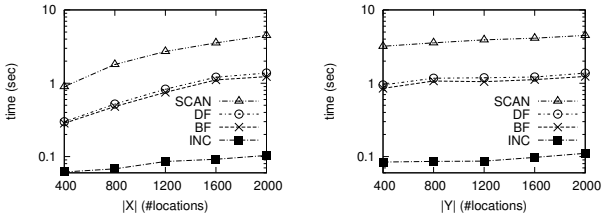
We can also see that INC has the lowest distance calculation cost. However, the performance improvement is much smaller than that of the traversal cost. This is because, INC has a higher distance calculation cost per traversed node than DF and BF. Specifically, DF and BF traverse the R-Tree R_Y of Y using the NN query, which computes the MINDIST for each child node encountered. For INC, on the other hand, traversal of R_Y (Algorithm 6) requires evaluation of lower bound and upper bound for each node encountered. As shown in Appendix B, each lower/upper bound calculation requires multiple MINDIST/MAXDIST calculations.



(a) Effect of $|X|$ (b) Effect of $|Y|$

Figure 7: Distance calculation costs of calculating HAUSDIST(X, Y)

Total execution time. Figure 8 displays the total execution times of the four methods. We can see that DF and BF provide an improvement of approximately 2.5 times over SCAN, and BF has a slightly better performance than DF (which conforms with the measure of distance calculation cost). INC is one order of magnitude faster than DF and BF.



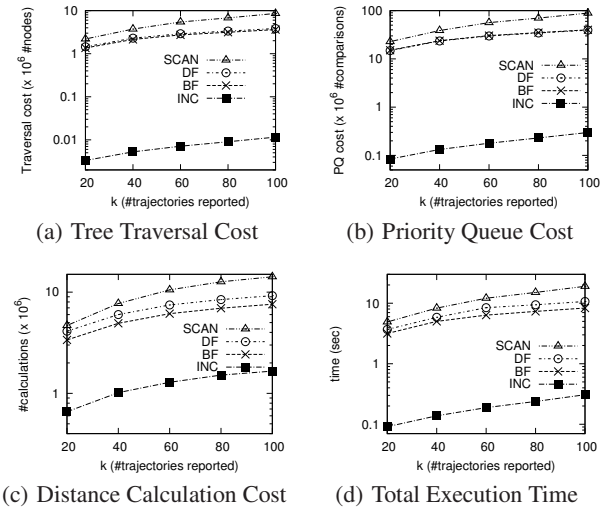
(a) Effect of $|X|$ (b) Effect of $|Y|$

Figure 8: Total execution times of calculating HAUSDIST(X, Y)

7.2 Incremental Trajectory Search

Results from the previous set of experiments have shown the effectiveness of the branch-and-bound methods (DF, BF and INC) when calculating HAUSDIST(X, Y) where X and Y are chosen at random. In this set of experiments, we compared the performance of the four methods as they were used to compute HAUSDIST in the incremental trajectory search algorithm (Lines 13 to 15 of Algorithm 8 in Appendix E). The search algorithm uses the HAUSDIST lower bound function as a search heuristic to estimate the distances from trajectories to a query trajectory Q. Hence, the trajectory selection is skewed towards those closer to Q.

Figure 9 displays results from the synthetic dataset (where each trajectory has 2000 locations) and the directionality of the HAUSDIST function is symmetrical (SYM). The results conform with the previous set of experiments (Section 7.1) in the following ways. First, DF and BF have similar cost improvements with respect to SCAN for tree traversal and priority queue costs. Second, BF performs slightly better than DF for the distance calculation cost and the total time. Finally, INC significantly outperforms all competitors in all cost measures.



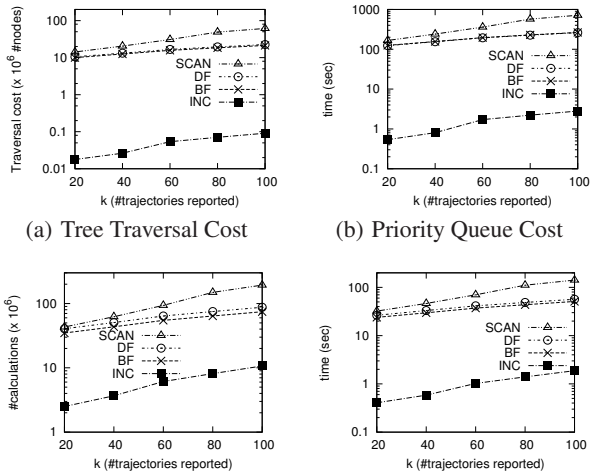
(a) Tree Traversal Cost (b) Priority Queue Cost

(c) Distance Calculation Cost (d) Total Execution Time

Figure 9: Effect of k on the synthetic dataset (SYM)

The main difference from the results of distance calculation experiments is that in the case of incremental trajectory search, only a marginal improvement is obtained from DF and BF for all cost measures. This result shows that the basic branch-and-bound methods (DF and BF) do not function well when calculating

HAUSDIST(X, Y) where X and Y are close to each other. The effectiveness of the incremental upper bound computation is demonstrated by the INC method, which retains an improvement factor similar to that for the distance calculation experiments (for all cost measures). Results for the real dataset are given in Figure 10. As can be seen, INC continues to significantly outperform the three competitors for all cost measures.



(a) Tree Traversal Cost (b) Priority Queue Cost

(c) Distance Calculation Cost (d) Total Execution Time

Figure 10: Effect of k on the real dataset (SYM)

Additional experimental results are given in Appendix H. These results conform with those given in this section.

7.3 Summary

In comparison to SCAN, the basic branch-and-bound methods (DF and BF) yield a reasonable cost improvement when just calculating the Hausdorff distance between two trajectories chosen at random. The improvement is marginal for incremental trajectory search. For INC, similar cost improvements in comparison to SCAN are obtained in all experiments and INC also outperforms all competitors for the four cost measures for all sets of experiments.

8. CONCLUDING REMARKS

We presented a novel technique to calculate the Hausdorff distance which utilizes hierarchical indexes. To calculate the Hausdorff distance HAUSDIST(X, Y), our technique traverses the indexes of X and Y in an incremental manner. We have compared our proposed method with (i) a baseline based upon an existing method [3] that scans an entire set X, and then performs indexed search over Y; (ii) two basic branch-and-bound algorithms (DF and BF) that considers Y as a single query object and traverses the index of X in a similar manner as the aggregate NN query [22]. Experiments show that our proposed method significantly outperforms its three competitors in terms of the traversal cost, priority queue maintenance cost, distance calculation cost and the total execution time.

Future work involves investigating the performance when calculating a partial Hausdorff distance in order to exclude outliers (as discussed in Section 5). We also want to extend our algorithm to handle trajectories represented as polylines (Appendix I).

Acknowledgements. This work was supported in part by the National Science Foundation under Grants IIS-09-48548, IIS-08-12377, CCF-08-30618, and IIS-07-13501.

9. REFERENCES

[1] P. K. Agarwal, S. Har-Peled, M. Sharir, and Y. Wang. Hausdorff distance under translation for points and balls. In *SCG*, pages 282–291, 2003.

[2] H. Alt, O. Aichholzer, and G. Rote. Matching shapes with a reference point. In *SCG*, pages 85–92, 1994.

[3] H. Alt, B. Behrends, and J. Blömer. Approximate matching of polygonal shapes (extended abstract). In *SCG*, pages 186–193, 1991.

[4] C. Andújar, P. Brunet, and D. Ayala. Topology-reducing surface simplification using a discrete solid representation. *ACM Trans. Graph.*, 21(2):88–105, 2002.

[5] W. G. Aref and I. F. Ilyas. An extensible index for spatial databases. In *SSDBM*, pages 49–58, 2001.

[6] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.

[7] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[8] N. Beckmann and B. Seeger. A revised R*-tree in comparison with related index structures. In *SIGMOD*, pages 799–812, 2009.

[9] D. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. *AAAI Workshop on Knowledge Discovery in Databases*, pages 229–248, 1994.

[10] L. Chen and R. Ng. On the marriage of l_p-norms and edit distance. *VLDB*, pages 792–803, 2004.

[11] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. *SIGMOD*, pages 491–502, 2005.

[12] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

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

[14] J. Hershberger and J. Snoeyink. Speeding up the douglas-peucker line-simplification algorithm. In *Intl. Symp. on Spatial Data Handling*, pages 134–143, 1992.

[15] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.

[16] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[17] D. P. Huttenlocher, K. Kedem, and J. M. Kleinberg. On dynamic voronoi diagrams and the minimum hausdorff distance for point sets under euclidean motion in the plane. In *SCG*, pages 110–119, 1992.

[18] D. P. Huttenlocher, G. A. Klanderman, and W. Rucklidge. Comparing images using the hausdorff distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(9):850–863, 1993.

[19] P. Indyk and S. Venkatasubramanian. Approximate congruence in nearly linear time. In *SODA*, pages 354–360, 2000.

[20] E. J. Keogh and C. Ratanamahatana. Exact indexing of dynamic time warping. *Knowl. Inf. Syst.*, 7(3):358–386, 2005.

[21] P.-F. Marteau. Time warp edit distance with stiffness adjustment for time series matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(2):306–318, 2009.

[22] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

[23] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[24] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.

[25] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD*, pages 343–354, 2000.

[26] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[27] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for KNN join processing. In *VLDB*, pages 756–767, 2004.

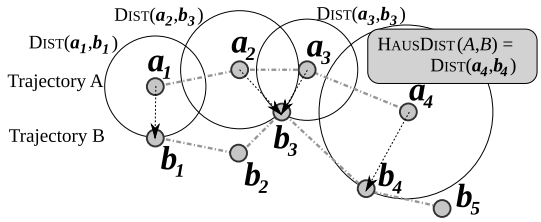
[28] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on gps data. In *UbiComp*, pages 312–321, 2008.

[29] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, pages 791–800, 2009.

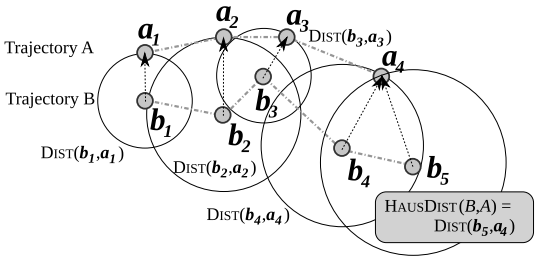
APPENDIX

A. HAUSDORFF DISTANCE EXAMPLES

Figure 11 illustrate the difference between the directed Hausdorff distance HAUSDIST(A, B) and HAUSDIST(B, A). The undirected Hausdorff distance SYMHAUSDIST(A, B) is given as the maximum of HAUSDIST(A, B) and HAUSDIST(B, A), which is HAUSDIST(B, A) in this example.



(a) HAUSDIST(A, B)



(b) HAUSDIST(B, A)

Figure 11: The Hausdorff distance (a) HAUSDIST(A, B) **evaluated as** MAX$\{$MINDIST$(\boldsymbol{a}, B) : \boldsymbol{a} \in A\}$ **and (b)** HAUSDIST(B, A) **evaluated as** MAX$\{$MINDIST$(\boldsymbol{b}, A) : \boldsymbol{b} \in B\}$**.**

B. UPPER BOUND AND LOWER BOUND

In this appendix, we derive the cost of upper bound and lower bound calculations by determining the number of MINDIST and MAXDIST calculations. In this cost measure, we assume that each of the following calculations costs one unit:

- MINDIST/MAXDIST from a point to an MBR;
- MINDIST/MAXDIST from an MBR to a point;
- MINDIST/MAXDIST from an MBR face to an MBR;
- MINDIST/MAXDIST from an MBR to an MBR face.

The HAUSDIST from a singleton set $\{\boldsymbol{p}\}$ to objects in an MBR B is the distance from $\boldsymbol{p}$ to the nearest object in B. As shown in Figure 12(a), a lower bound of HAUSDIST from a point $\boldsymbol{p}$ to objects in an MBR B can be computed as the smallest possible distance from $\boldsymbol{p}$ to the nearest object in B. That is, HAUSDISTLB$(\boldsymbol{p}, B) =$

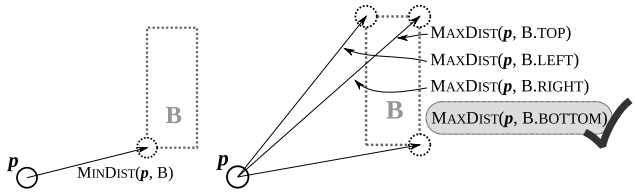
$$\text{MINDIST}(\boldsymbol{p}, B).$$

Hence, this operation requires 1 MINDIST calculation.

As shown in Figure 12(b), an upper bound of HAUSDIST from a point $\boldsymbol{p}$ to objects in an MBR B can be computed as the greatest distance from $\boldsymbol{p}$ to its nearest object in B. In this case, we use the MAXNEARESTDIST [24] as an upper bound. Specifically, we exploit the fact that each of the four faces (B.TOP, B.BOTTOM, B.LEFT and B.RIGHT) has at least one object that touches it. We then compute the worst case of each of the faces and the upper bound is obtained as the overall minimum. That is, HAUSDISTUB$(\boldsymbol{p}, B) =$

$$\text{MIN}\{\text{MAXDIST}(\boldsymbol{p}, f_b) : f_b \in \text{FacesOf}(B)\}.$$

This operation requires 4 MAXDIST calculations.



(a) Lower bound (b) Upper bound

Figure 12: Lower bound and upper bound from a point $\boldsymbol{p}$ to an MBR B

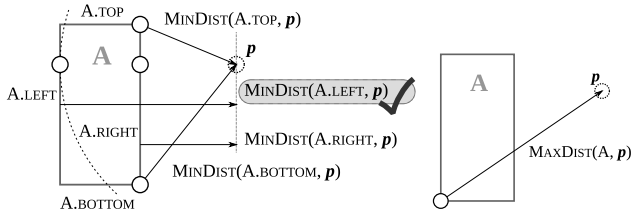
The HAUSDIST from objects in an MBR A to a singleton set $\{\boldsymbol{p}\}$ is the distance to $\boldsymbol{p}$ from its farthest object in the MBR. Figure 13(a) shows a lower bound computed as the minimum distance of this farthest object, MINFARTHESTDIST [24]. That is, HAUSDISTLB$(A, \boldsymbol{p}) =$

$$\text{MAX}\{\text{MINDIST}(f_a, \boldsymbol{p}) : f_a \in \text{FacesOf}(A)\}.$$

Hence, this operation requires 4 MINDIST calculations. Similarly, Figure 13(b) shows an upper bound computed as the maximum distance of the farthest object with respect to $\boldsymbol{p}$. That is, HAUSDISTUB$(A, \boldsymbol{p}) =$

$$\text{MAXDIST}(A, \boldsymbol{p}).$$

Hence, this operation requires 1 MAXDIST calculation.



(a) Lower bound (b) Upper bound

Figure 13: Lower bound and upper bound from an MBR A to point $\boldsymbol{p}$

We now determine the calculation costs for the case where both entries A and B are MBRs. As shown in Figure 14, we reuse the examples given in Section 3. In this case, a lower bound can be computed by breaking down both A and B into faces as

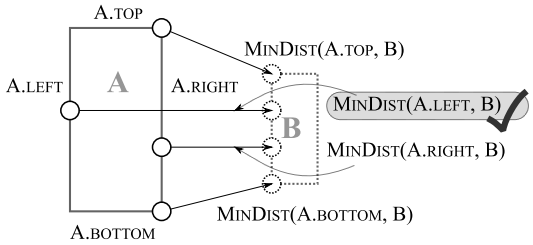
$$\text{HAUSDISTLB}(A, B) = \text{MAX}\{\ \text{MIN}\{\text{MINDIST}(f_a, f_b) : f_b \in \text{FACESOF}(B)\} : f_a \in \text{FACESOF}(A)\}.$$

Since MIN$\{$MINDIST$(f_a, f_b) : f_b \in$ FACESOF$(B)\}$ is equivalent to MINDIST(f_a, B), this operation requires 4 MINDIST calculations. Similarly, an upper bound can be evaluated as

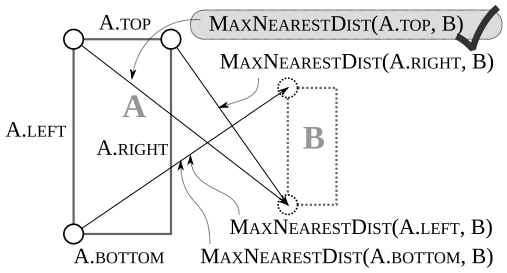
$$\text{HAUSDISTUB}(A, B) = \text{MAX}\{\ \text{MIN}\{\text{MAXDIST}(f_a, f_b) : f_b \in \text{FACESOF}(B)\} : f_a \in \text{FACESOF}(A)\}.$$

Note that MIN$\{$MAXDIST$(f_a, f_b) : f_b \in$ FACESOF$(B)\}$ can be rewritten as MAXNEARESTDIST(f_a, B). However, evaluation of MAXNEARESTDIST also requires 4 MAXDIST calculations. Hence, this operation requires 16 MAXDIST calculations in total.

Summary of distance calculation costs in the number of MINDIST and MAXDIST calculations is given in Table 1.



(a) Lower bound



(b) Upper bound

Figure 14: Lower bound and upper bound an MBR A to another MBR B

Table 1: Summary of calculation costs

	Point to MBR	MBR to Point	MBR to MBR
Lower bound	1	4	4
Upper bound	4	1	16

C. CORRECTNESS OF UPPER BOUND CALCULATED FROM SUB-NODES

We verify the correctness of using the minimum upper bound of child nodes as an overall upper bound of the parent as follows.

LEMMA 1. *Given two sets of objects X and Y which are enclosed by two R-Tree nodes A and B, respectively. Let $\{B_1, ..., B_n\}$ denote the set of sub-nodes in B. The value*

$$\text{MIN}\{\text{HAUSDISTUB}(A, B_i) : B_i \in \{B_1, ..., B_n\}\}$$

is an upper bound of $\text{HAUSDIST}(X, Y)$.

PROOF. We prove this lemma by showing that for all B_i in $\{B_1, ..., B_n\}$, $\text{HAUSDIST}(X, Y)$ cannot be greater than $\text{HAUSDISTUB}(A, B_i)$.

Assume that the set Y is decomposed into subsets $Y_1, ..., Y_n$ where each is enclosed by $B_1, ..., B_n$ respectively. Hence, for all i in $[1, ..., n]$,

$$\text{HAUSDIST}(X, Y_i) \leq \text{HAUSDISTUB}(A, B_i).$$

Since each Y_i is a subset of Y, a closer element to X could exist in Y. We have $\text{HAUSDIST}(X, Y) \leq \text{HAUSDIST}(X, Y_i)$.

For all B_i in $\{B_1, ..., B_n\}$,

$$\text{HAUSDIST}(X, Y) \leq \text{HAUSDISTUB}(A, B_i).$$

As a result, the minimum value of $\text{HAUSDISTUB}(A, B_i)$ over the set $\{B_1, ..., B_n\}$ is an upper bound of $\text{HAUSDIST}(X, Y)$. □

D. CALCULATING THE UNDIRECTED HAUSDORFF DISTANCE

The undirected Hausdorff distance function $\text{SYMHAUSDIST}(X, Y)$ is defined as the maximum between $\text{HAUSDIST}(X, Y)$ and $\text{HAUSDIST}(Y, X)$. Hence, both $\text{HAUSDIST}(X, Y)$ and $\text{HAUSDIST}(Y, X)$ can be used as lower bounds for $\text{SYMHAUSDIST}(X, Y)$. We can exploit this observation to speed up the computation of $\text{SYMHAUSDIST}(X, Y)$ by (i) evaluating the direction that has a greater lower bound first, and (ii) use the result as a lower bound for the other direction. Specifically, given two point sets X and Y, $\text{SYMHAUSDIST}(X, Y)$ can be calculated as follows:

- calculate a lower bound L_1 of $\text{HAUSDIST}(X, Y)$ using the MBRs of the two sets;
- calculate a lower bound L_2 of $\text{HAUSDIST}(Y, X)$ using the MBRs of the two sets;
- if L_1 is greater than L_2, then let S_1 be X and S_2 be Y;
- otherwise let S_1 be Y and S_2 be X;
- calculate d_1 as Inc-$\text{HAUSDIST}(S_1, S_2)$;
- calculate d_2 as Inc-$\text{HAUSDIST}^*(S_2, S_1, d_1)$ (Algorithm 7);
- return the maximum of d_1 and d_2 as the resultant HAUSDIST.

Inc-$\text{HAUSDIST}^*()$ is a slight modification to the proposed algorithm (Algorithm 4). The algorithm accepts the distance d which is the HAUSDIST previously computed in the opposite direction as a lower bound. The algorithm terminates upon discovery of an upper bound *UB* that is smaller than d (Lines 9 to 10). This is because, further computation would yield a distance smaller than what we already have and cannot affect the overall maximum distance. The same principle can also be applied to DF-HAUSDIST (Algorithm 2) and BF-HAUSDIST (Algorithm 3).

Algorithm 7: Inc-$\text{HAUSDIST}^*(X, Y, d)$

input : Point Set X, Point Set Y, Distance d
output: Maximum of d and the Hausdorff Distance from X to Y

```
1  RTree R_X ← Create an R-Tree for X;
2  RTree R_Y ← Create an R-Tree for Y;
3  MainPQ MPQ ← Create a "descending order" PQ;
4  SecPQ SPQ ← Create an "ascending order" PQ;
5  Insert((RootOf(R_Y), 0), SPQ);
6  Insert((RootOf(R_X), 0, SPQ), MPQ);
7  while MainPQ is not empty do
8      MainPQ-Entry (N_X, UB, SPQ) ← Dequeue(MPQ);
9      if UB ≤ d then
10         return d;
11     SecPQ-Entry (N_Y, LB) ← Head of SPQ;
12     if N_X and N_Y are both points then
13         return UB;
14     else if N_X is in a shallower depth than N_Y then
15         TraverseX(N_X, SPQ, MPQ)
16     else
17         TraverseY(N_X, UB, SPQ, MPQ)
```

E. TRAJECTORY SEARCH ALGORITHM

Algorithm 8 applies the concept of incremental nearest neighbor search [16] and proceeds as follows. The initialization steps (Lines 1 to 4) include (i) creating an MBR M_Q for Q; (ii) creating an R-Tree R for all trajectories in $\mathcal{D}$; (iii) initializing a priority queue PQ; (iv) inserting a priority queue entry with RootOf(R) into PQ. The best-first traversal is controlled by the while loop (Lines 5 to 20), which explores nodes in R according to the HAUSDIST estimator (given in Algorithm 9).

Algorithm 8: TRAJSEARCH(Q, $\mathcal{D}$, T)

input : Query trajectory Q, Trajectory set $\mathcal{D}$ and Distance direction T (FROM-Q, TO-Q or SYM)
output: Trajectory with the smallest HAUSDIST with respect to Q

```
1  M_Q ← Create an MBR of Q;
2  R ← Create an R-Tree for D;
3  Priority Queue PQ ← Create an "ascending order" PQ;
4  Insert((RootOf(R), 0, False), PQ);
5  while PQ is not empty do
6      PQ-Entry (Node N, Distance d, Bool IsFinal) ← Dequeue(PQ);
7      if N contains one trajectory then
8          if IsFinal then
9              return N;
10         else
11             Distance d;
12             switch the value of T do
13                 case FROM-Q: d ← HausDist(Q, N);
14                 case TO-Q: d ← HausDist(N, Q);
15                 case SYM: d ← MAX {HausDist(Q, N),
                                      HausDist(N, Q)};
16             Insert((N, d, True), PQ);
17     else
18         for each Child C of N do
19             Distance d ← HausEst(M_Q, C, T);
20             Insert((C, d, False), PQ);
```

The Hausdorff distance of a query MBR and a node containing trajectories can be estimated as follows. As input, the algorithm accepts (i) an MBR M_Q corresponding to the query trajectory Q, (ii) a node C in the R-Tree containing a set of possible trajectories, and (iii) the direction of the distance function (FROM-Q, TO-Q or SYM). The output of the algorithm is a lower bound on the HAUSDIST for all trajectories in C with respect to Q. If C corresponds to one trajectory, then the estimate is described as follows.

- HAUSDISTLB(M_Q, C) for FROM-Q.
- HAUSDISTLB(C, M_Q) for TO-Q.
- MAX{HAUSDISTLB(M_Q, C), HAUSDISTLB(C, M_Q)} for SYM.

Otherwise, if C contains multiple trajectories (C is an index node), an estimate can be computed as follows.

- FROM-Q — HAUSDISTLB(M_Q, C). Since M_Q is the MBR of Q, it is safe to assume that each face of M_Q has at least one data point. Please refer to Figure 14(a).
- TO-Q — MINDIST(C, M_Q). Since C contains multiple trajectories, the nearest trajectory can be anywhere in C. It is safe to assume only that the nearest trajectory cannot be outside C and use MINDIST as the lower bound.
- SYM. This is evaluated as the maximum of the first two estimates. However, since MINDIST(C, M_Q) cannot be greater than HAUSDISTLB(M_Q, C), HAUSDISTLB(M_Q, C) is always the maximum of the two and there is no need to calculate MINDIST(C, M_Q).

F. EXAMPLE SEARCH RESULTS

This appendix provides example trajectory search results which are ranked according to the HAUSDIST from a query trajectory Q using the city of Oldenburg dataset (described and used in the experimental studies, Section 7). Figure 15 shows results of ranks $[1, 51, ..., 451]$ where each is accompanied by the distance from Q. We can see that as the rank and distance increase the deviation from Q also increases. These search results illustrate the effectiveness of the Hausdorff distance as a similarity measure for trajectories.

Algorithm 9: HAUSEST(M_Q, C, T)

input : Query MBR M_Q, Node C and Direction T (FROM-Q, TO-Q or SYM)
output: Optimistic Estimate of HAUSDIST

```
1  Distance L;
2  if C contains one trajectory then
3      switch the value of T do
4          case FROM-Q: L ← HausDistLB(M_Q, C);
5          case TO-Q: L ← HausDistLB(C, M_Q);
6          case SYM: L ← MAX {HausDistLB(M_Q, C),
                             HausDistLB(C, M_Q)};
7  else
8      switch the value of T do
9          case FROM-Q: L ← HausDistLB(M_Q, C);
10         case TO-Q: L ← MinDist(C, M_Q);
11         case SYM: L ← HausDistLB(M_Q, C);
12 return L
```

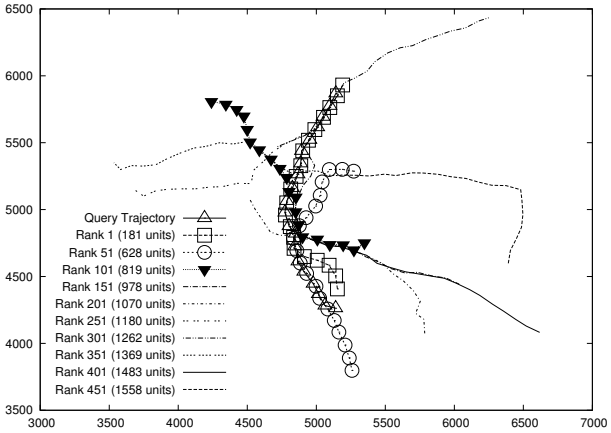


Figure 15: Similarity search results ranked according to HAUSDIST from the query trajectory Q

G. EFFECTIVENESS OF TRAJSEARCH

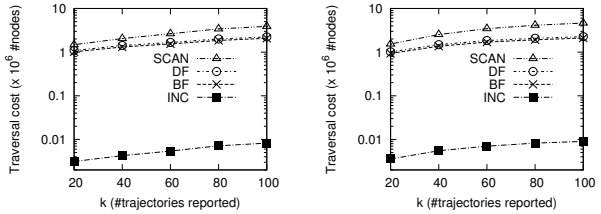
Table 2 shows the proportion of trajectories encountered by TRAJSEARCH. As we can see, in the case of the Oldenburg dataset with a k value of 20, our search algorithm considers only 42 trajectories out of a collection of 2000. In other words, in order to retrieve the 20 most similar trajectories based on the symmetrical Hausdorff distance, our algorithm considers only 22 non-resultant trajectories. The number of encountered trajectories increases as k increases. When k is 100 (i.e., ranking 5% of the Oldenburg dataset and 3.7% of the GeoLife dataset), TRAJSEARCH accesses only 8.3% and 13.6% of the trajectories, respectively.

Table 2: Proportion of Trajectories Accessed

	Oldenburg (2000 Trajectories)		GeoLife (2669 Trajectories)	
k	Number of Trajectories	Percentage	Number of Trajectories	Percentage
20	42	2.1%	65	2.4%
40	72	3.6%	94	3.5%
60	106	5.3%	138	5.1%
80	131	6.6%	202	7.6%
100	166	8.3%	363	13.6%

H. ADDITIONAL EXPERIMENTAL RESULTS

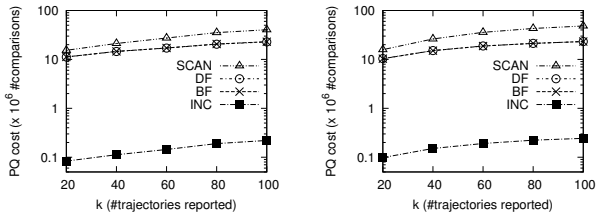
In addition to the results for the symmetric Hausdorff distance (SYM) given in Section 7, we report experimental results of using the Hausdorff distance to find trajectories that minimize the distance in the directions of "FROM-Q" and "TO-Q". Figure 16 compares the traversal costs of the four methods. We see that the traversal costs of DF and BF are slightly lower than that of SCAN. Significant improvement of over an order of magnitude was obtained by INC. This result confirms our hypothesis that a more effective upper bound for HAUSDIST(X, Y) can be obtained by examining deeper nodes in R_Y instead of just the root node.



(a) Directed (FROM-Q) (b) Directed (TO-Q)

Figure 16: Effect of k on the traversal costs

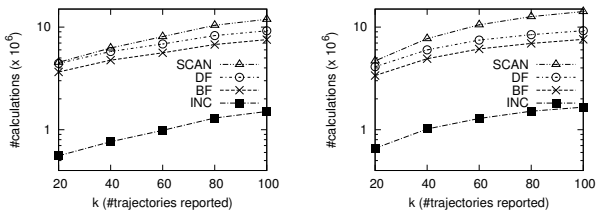
Figure 17 shows the comparison of the four methods in terms of priority queue cost. The results conform with those of the traversal cost. DF and BF incur similar costs and perform marginally better than SCAN. INC significantly outperforms all competitors.



(a) Directed (FROM-Q) (b) Directed (TO-Q)

Figure 17: Effect of k on the priority queue costs

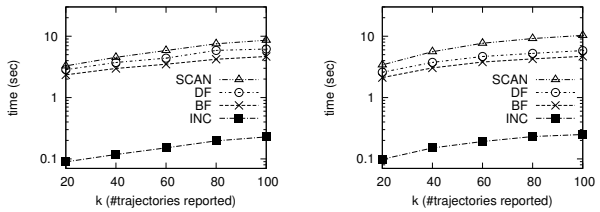
Figure 18 shows the distance calculation costs of the four methods. In this cost measure, we can see that DF has a slightly higher cost than BF. This is because, DF computes lower bounds of nodes for pruning purposes (Line 7, Algorithm 2), while this step is not required for BF (Algorithm 3) since pruning is handled implicitly by the best-first traversal order. We can also see that INC continues to outperform all other methods.



(a) Directed (FROM-Q) (b) Directed (TO-Q)

Figure 18: Effect of k on the distance calculation costs

Figure 19 displays the total execution times of the four methods. We can see that DF and BF provide a marginal speed improvement over SCAN. The performance difference between DF and BF is due to the difference in the distance calculation cost. INC, which is the method that has the lowest traversal cost, priority queue cost and distance calculations, also significantly outperforms all other methods in this cost measure.



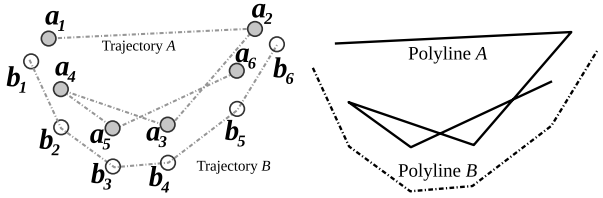
(a) Directed (FROM-Q) (b) Directed (TO-Q)

Figure 19: Effect of k on the total execution times

As can be seen from the synthetic dataset, results FROM-Q and TO-Q given in this section are similar to that for SYM given in Section 7.2. Hence, we omit presentation of results from FROM-Q and TO-Q for the real dataset.

I. ALTERNATIVE TRAJECTORY REPRESENTATION

Figure 20(a) illustrates a problem that may arise when using HAUSDIST to determine the similarity between two trajectories A and B. The figure shows two trajectories A and B represented as two point sets $\{\boldsymbol{a_1}, ..., \boldsymbol{a_6}\}$ and $\{\boldsymbol{b_1}, ..., \boldsymbol{b_6}\}$, respectively. Since every point in A is close to at least one point in B, HAUSDIST(A, B) yields a distance which is not indicative of the actual discrepancy of the trajectory A with respect to B. One approach to mitigate this problem is to ensure that the frequency in which trajectories are sampled is sufficient to preserve their continuity. As a result, points that are temporally close to each other are also spatially close to each other.



(a) Lower sampling rate (b) Polyline Representation

Figure 20: Discrete and continuous trajectory representations

A more sophisticated approach to addressing this problem is to represent trajectories as piecewise linear curves as illustrated by Figure 20(b). In this case, the Hausdorff distance from Trajectory A to Trajectory B is given by a point along Polyline A that maximizes the minimum distance to Polyline B. Another benefit of this method is that it allows line simplification techniques to be applied to trajectories [14]. As a result, consecutive locations that are approximately co-linear are simplified to a single line segment reducing the number of line segments to store and to process.

Our proposed Hausdorff distance computation algorithm (Algorithm 4) can be adjusted to handle this extension. Specifically, we can replace calculation of the Hausdorff distance between two points $\boldsymbol{a}$ and $\boldsymbol{b}$ (which is simply DIST$(\boldsymbol{a}, \boldsymbol{b})$) by the Hausdorff distance from one line segment to another. Details on how this calculation can be derived is given by Alt et al. [3]. This extension will be investigated as part of our future research.