Identifying K Primary Corridors from urban bicycle GPS trajectories on a road network

Zhe Jiang *, Michael Evans, Dev Oliver, Shashi Shekhar

Department of Computer Science and Engineering, University of Minnesota, Twin Cities, Minneapolis, United States

Abstract

Given a set of GPS tracks on a road network and a number k, the K-Primary-Corridor (KPC) problem aims to identify k tracks as primary corridors such that the overall distance from all tracks to their closest primary corridors is minimized. The KPC problem is important to domains such as transportation services interested in finding primary corridors for public transportation or greener travel (e.g., bicycling) by leveraging emerging GPS trajectory datasets. However, the problem is challenging due to the large amount of shortest path distance computations across tracks. Related trajectory mining approaches, e.g., density or frequency based hot-routes, focus on anomaly detection rather than identifying representative corridors minimizing total distances from other tracks, and thus may not be effective for the KPC problem. Our recent work proposed a k-Primary Corridor algorithm that precomputes a column-wise lookup table of network Hausdorff distances. This paper extends our recent work with a new computational algorithm based on lower bound filtering. We design lower bounds of network Hausdorff distances based on the concept of track envelopes and propose three different track envelope formation strategies based on random selection, overlap, and Jaccard coefficient respectively. Theoretical analysis on proof of correctness as well as computational cost models are provided. Extensive experiments and case studies show that our new algorithm with lower bound filtering significantly reduces the computational time of our previous algorithm, and can help effectively determine primary bicycle corridors.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Given a set of GPS tracks on a road network and a number k, the K-Primary-Corridor (KPC) problem aims to identify k tracks as primary corridors such that the overall distance of every track to its closest primary corridors is minimized. Fig. 1 is a real world example, where Fig. 1(a) shows a collection of bicycle GPS tracks in Minneapolis, and Fig. 1(b) shows eight primary corridors identified.

Application: KPC is a trajectory mining problem in the field of urban computing [27], which studies the acquisition, integration, and analysis of urban data to address major issues that modern cities face. The KPC problem is important to a variety of domains, such as transportation services interested in finding primary corridors for public transportation or greener travel (e.g., bicycling) by leveraging emerging GPS trajectory datasets. For example, geographers from the University of Minnesota study urban cyclist behaviors related to route choices and safety via the cyclists’ GPS tracks [10]. In the application of urban bicycle corridor planning, investment in primary corridors can facilitate cyclists and encourage use. The reason why primary corridors are defined as “medoids” is that we want to...
minimize the cost of bicyclists shifting from their current tracks to primary corridors so that the use of these primary corridors are encouraged. Investment in infrastructure on primary corridors can facilitate safe and efficient bicycle travel, and has been shown to have numerous societal benefits, such as reduced greenhouse gas emissions, energy consumption, as well as healthcare costs [18].

**Challenges:** The KPC problem is challenging due to the large number of shortest path computations on a road network. First, computing distance across two GPS tracks is expensive on the network space. Consider the popular Hausdorff distance as an example. When applied to a network space [22], it measures the maximum shortest path distance from every node in the source track to the target track (i.e., the upper bound of the transition cost). A brute-force method calculates the distance from every node in the source track to the closest node in the target track via a shortest-path algorithm, e.g., Dijkstra [5]. Furthermore, in order to select \( k \) primary corridors, distances across almost all pairs of tracks need to be compared, the number of which is the square of the number of tracks. For example, given 1000 tracks, a brute-force method needs to compute 1,000,000 Hausdorff distances, each of which may need several Dijkstra calls.

**Related work:** Fig. 2 summarizes related trajectory mining techniques in the existing literature, which can be categorized into two groups, geometry space based and network space based. Techniques in geometry space use Euclidean distances [23,1,16], and thus cannot model network travel costs (e.g., roads along two sides of a river are close in geometry space but distant in network space). Techniques in network space include “hot” or popular route approaches, such as density-based [17,14,4], frequency-based [15] and hierarchical clustering [26,22]. More specifically, a method [14] similar to DBScan [6] finds dense road segments (i.e., with high number of trajectories) and merges them into dense routes. A density-based algorithm called FlowScan [17] propose to identify hot routes based on a definition of “traffic density reachable” (i.e., the number of trajectories passing two road segments is higher than a threshold). Another proposed technique [4] identifies popular routes from source locations to target locations based on high transitive probability. A frequency-based approach [15] uses Apriori algorithms to find road network sub-paths whose supports are higher than a threshold. In summary, all of these “hot” route discovery techniques are based on finding track anomalies (e.g., high-frequency or high density tracks).

In contrast, our recent work [7,8] focuses on identifying representative tracks, namely primary corridors. The overall distance from all tracks to their closest primary corridors is minimized. We use directed network Hausdorff distance to measure (dis)similarity across tracks. Hausdorff distance was first used to measure similarity between two geometric objects (e.g., polygons, lines, sets of points) [12], and has been shown to be a useful tool in geometric space for measuring similarity between trajectories [11,19,3,2]. Recently, it has been generalized to measure similarity between trajectories in network space [22,9]. We use network Hausdorff distance in our problem instead of geometry Hausdorff distance because the latter cannot model real travel cost in a road network (e.g., two parallel tracks on opposite sides of a river have a small geometry distance but a large actual network distance). Edit distance is not used because it cannot distinguish different levels of distance between non-overlapping tracks (i.e., edit distance is zero for all non-overlapping tracks no matter how far they are separated).

In our recent work [7,8], we (1) formally defined a new K-Primary-Corridor problem; (2) proposed a computational algorithm (Urban2013 approach) based on a column-wise lookup table that reduces the computational cost of a brute-
force approach; and (3) provided a case study on real world bicycle GPS trajectories in Minneapolis, MN.

New contributions: This paper reorganizes and extends our previous papers \cite{7,8} with the following additional contributions:

1. We propose a novel computational algorithm with lower bound filtering based on track envelopes.
2. We propose three track envelope formation strategies, i.e., random envelopes, overlap-based envelopes, and Jaccard coefficient-based envelopes, and analyze factors influencing the tightness of lower bounds.
3. We prove the correctness of the new algorithm and provide theoretical analysis on the computational cost models.
4. We present extensive experimental evaluations comparing the computational performance of our new algorithm with our previous algorithm.
5. We offer in the case study on real world urban cyclists’ GPS trajectories in Minneapolis to compare our \(k\)-Primary-Corridor algorithms with handcrafted primary corridors by geographers as well as density-based hot routes produced by a related work.

Scope: In this paper, we identify primary corridors based on Hausdorff distance on the network space. Other distances such as geometry Hausdorff distance and edit distance \cite{26,14} are not considered. The choice of \(k\) in the input is determined by users and is beyond our scope. We use Dijkstra \cite{5} for shortest-path distance computation. Other methods (e.g., Floyd–Warshall \cite{5}) are not used. Finally, since our focus is on the spatial footprint of GPS tracks, we ignore information of time, sequential order, and directions in GPS tracks in our study.

Organization: The paper is organized as follows: Section 2 introduces basic concepts and formalizes the K-Primary-Corridor problem. Section 3 describes computational algorithms to solve the problem, including our previous algorithms and the new one proposed in this paper. Section 4 provides theoretical analysis on algorithm correctness and computational cost models. Experimental evaluation and a real world case study are presented in Section 5. Section 6 discusses our approach and some other relevant work. Section 7 concludes the paper with future work.

2. Basic concepts and problem statement

This section first introduces basic concepts. Then, it formally defines the K-Primary-Corridor problem and illustrates it.

2.1. Basic concepts

We now introduce the following basic concepts.

Road network: A road network is a graph \(G\) whose nodes represent road intersections and whose edges represent road segments between intersections. The weight of an edge measures its travel distance (or time cost). For simplicity, we assume that the graph is undirected. An example of a road network can be found as the grey grid in Fig. 3(a), which has \(7 \times 7 = 49\) nodes and 84 edges of unit lengths.

GPS track: A GPS track is a sequence of GPS locations recorded during a trip. On a road network, a GPS track can be represented as a sequence of nodes, or a path traversed during a trip. We ignore the time, direction, and sequential order of a track because we are only interested in the underlying route choices (spatial footprint of tracks) in applications such as urban bicycle corridor planning. Thus, a track \(t\) can be formally defined as a set \((v_1, v_2, \ldots, v_k)\), where \(v_i\) is a node in a road network graph, and \(L\) is the length of the track. For instance, the blue line \((t_3)\) in Fig. 3(a) is a track of length 8.

Directed network Hausdorff distance: Directed network Hausdorff distance is defined as

\[
H(t_i, t_j) = \max_{v_i \in t_i} \min_{v_j \in t_j} d(v_i, v_j),
\]

where \(H(t_i, t_j)\) is the directed network Hausdorff distance from a source track \(t_i\) to a target track \(t_j\), \(v_i\) is a node in the track \(t_i\), \(v_j\) is a node in the track \(t_j\), and \(d\) is the shortest path distance. For example, in Fig. 3(a), \(H(t_4, t_1) = 2\), since the longest shortest path distance from nodes in \(t_4\) to \(t_1\) is 2 (from the rightmost node on \(t_4\)).
Similarly, we can find that $H(t_1, t_4) = 1$. For simplicity, we use Hausdorff distance to refer to directed network Hausdorff distance in the remainder of the paper. Note that this Hausdorff distance definition is asymmetric, unlike the Hausdorff distance function (or metric) in mathematics. The intuition behind the definition is that it measures the

![diagram](image)

**Fig. 4.** A running example of K-Primary-Corridor approach in general. (a)
upper bound of the travel distance from one track to another.

**Primary corridor**: The primary corridor (PC) of a set of GPS tracks is defined as the track that has the minimum overall Hausdorff distance from all other tracks, similar to medoids in the K-Medoids clustering problem [21]. In other words, a primary corridor is a track such that the total travel distance is minimized for changing routes from current tracks to their closest primary corridors. For example, the primary corridor of tracks \( t_1, t_2, t_3, t_4 \), whose overall distance is 3, is \( t_1 \). In the application of urban bicycle corridor planning, investment in primary corridors can facilitate cyclists and encourage use. The reason why primary corridors are defined as “medoids” is that we want to minimize the cost of bicyclists shifting from their current routes to primary corridors so that the use of these primary corridors are encouraged.

### 2.2. Problem definition

Based on the concepts above, the K-Primary-Corridor problem is formally defined as follows:

**Given**:
- a road network as a graph \( G(V, E) \)
- a set of GPS tracks \( T \) on \( G \)
- a number \( k \)

**Find**:
- \( k \) primary corridors

**Objective**:
- minimize sum of distances from every track to its closest PC

**Constraints**:
- primary corridors are tracks within \( T \)
- \( G \) is connected, undirected, and with non-negative edges
- directed network Hausdorff distance is used

**Problem description**: The problem aims to select \( k \) primary corridors (PCs) from a set of tracks. The objective is to minimize the total sum of distances from tracks to their closest PCs. The output \( k \) primary corridors minimize the total travel distance for changing routes from all other tracks to their closest PCs.

**Example**: Fig. 3(a) and (b) shows examples of problem inputs and outputs. The inputs include a road network depicted by an underlying grid with 49 nodes and 84 edges (of unit length), ten GPS tracks from \( t_1 \) to \( t_{10} \) shown as solid lines in different colors, and a \( k=2 \). The two hottest tracks (maximizing density or frequency), \( t_5 \) and \( t_{10} \), are concentrated on the right part of the network. These two tracks have high overall distance from other tracks, and cannot serve cyclists on the left side of the map. In contrast, the two output primary corridors, \( t_1 \) and \( t_5 \), have small overall travel distance from other tracks (both have a sum of distances as 3), and can serve cyclists both on the left side and on the right side.

### 3. Proposed approach

We present an overview of our computational approach to the K-Primary-Corridor problem, followed by descriptions of a brute-force algorithm and our previous algorithm based on a row-wise lookup table [7,8]. We then propose a new computational algorithm based on lower bound filtering.

#### 3.1. Approach overview

Our approach to identifying \( k \) primary corridors consists of iterative steps similar to the k-medoids clustering algorithm [13,21]. Primary corridors are “medoids”.

The approach begins by initializing the \( k \) primary corridors with \( k \) randomly selected tracks, and then performs iterative operations. Each iteration has two phases: **track assignment** and **primary corridor updating**. The track assignment phase fixes the current primary corridors, and assigns each track to its closest primary corridor by Hausdorff distance. In this way, tracks are grouped into \( k \) clusters (or groups). The primary corridor updating phase fixes the \( k \) clusters of tracks, and updates the primary corridor of each cluster. More specifically, within every subset, it selects the track to which the total Hausdorff distances from the remaining tracks is the minimum, and makes it a new primary corridor. The iterations keep running until the \( k \) primary corridors no longer change in the updating phase.

A running example: Fig. 4 illustrates the execution process. The inputs are the same as the problem example in Fig. 3. There are ten tracks (\( t_1 \) to \( t_{10} \), \( k=2 \), and a set of two initial primary corridors \( t_1, t_3 \). In the first iteration, the algorithm fixes these two primary corridors, and assigns each track to its closest primary corridor. Tracks \( t_1 \) and \( t_2 \) are assigned to the first cluster, and new tracks are assigned to the second cluster, as shown in Fig. 4(a). The algorithm then updates the primary corridor of each cluster. According to the sum of the distances from the remaining tracks in the cluster (listed in Fig. 4(b)), the new primary corridors are \( t_1 \) and \( t_5 \). The second iteration runs similarly. Fig. 4(c) and (d) shows new track assignments (\( t_3 \) and \( t_4 \) are assigned to the first cluster now), and new primary corridors \( t_1 \) and \( t_5 \). Since the primary corridors stay unchanged, the algorithm converges and terminates. The output primary corridors are \( t_1 \) and \( t_5 \).

#### 3.2. Brute-force computational algorithm

A brute-force method (Algorithm 1) for finding \( k \) primary corridors computes a Hausdorff distance on the fly whenever its value is needed. **Track assignment** computes the Hausdorff distances from every track to every primary corridor, and finds the primary corridor with the minimum distance. **Primary corridor updating** evaluates every track in every trace group as a candidate new primary corridor. The sum of Hausdorff distances from the remaining tracks is computed in order to compare the different candidates.

For each Hausdorff distance, the brute-force algorithm computes the shortest path distance from every node in the source track to every node in the target track. Several Dijkstra calls are needed, each of which uses a
different node in the source track as the source. Then the Hausdorff distance is computed as the maximum of these distances.

Algorithm 1. K-Primary-Corridor-Brute-Force \((G, T, k)\).

**Input:**
- \(G\): an undirected graph representing a road network
- \(T\): a set of GPS tracks on the road network
- \(k\): the number of primary corridors

**Output:**
- \(k\) primary corridors

1: initialize \(k\) primary corridors \(PC = \{pc_j | j = 1, \ldots, k\}\)
2: while \(PC\) is not updated do
3: for each track \(t_i\) in \(T\) do
4: \(//assign \(t_i\) to its closest primary corridor \(pc_j\)
5: for each \(pc_j \in PC\) do
6: compute Hausdorff distance \(H(t_i, pc_j)\)
7: find the \(pc_j\) minimizing \(H(t_i, pc_j)\)
8: assign \(t_i\) to the \(pc_j\) (i.e., cluster \(T_j\) \(\rightarrow\) \(T_j\) \(\cup\) \(\{t_i\}\))
9: for each track cluster \(T_j\) do
10: \(//update its new primary corridor \(pc_j\)
11: for each candidate track \(pc_{0j} \in T_j\) do
12: sum \(\leftarrow 0\)
13: for each remaining track \(t_j \in T_j\) \(\setminus\) \(\{pc_{0j}\}\) do
14: compute Hausdorff distance \(H(t_j, pc_{0j})\)
15: sum \(\leftarrow\) sum + \(H(t_j, pc_{0j})\)
16: \(pc_j \leftarrow pc_{0j}\) minimizing sum
17: Return \(PC = \{pc_j | j = 1, \ldots, k\}\)

3.3. Lookup table approach

The brute-force algorithm has significant computational overhead in the primary corridor updating phase. It computes Hausdorff distances across almost all pairs of tracks, each of which requires several Dijkstra calls. To reduce the computational overhead, we recently proposed a column-wise lookup table approach (named the Table Lookup approach) [8]. The only difference between the Table Lookup approach and the brute-force approach is that the former pre-computes a matrix of Hausdorff distances between each pair of tracks (also called the lookup table). In the brute-force approach, each Hausdorff distance must be computed via one or more Dijkstra call; while Table Lookup allows each column of Hausdorff distances to be computed via only one Dijkstra call.

Algorithm 2. K-Primary-Corridor-LookupTable\((G, T, k)\).

**Input:**
- \(G\): an undirected graph representing a road network
- \(T\): a set of GPS tracks on the road network
- \(k\): the number of primary corridors

**Output:**
- \(k\) primary corridors

1: initialize \(k\) primary corridors \(PC = \{pc_j | j = 1, \ldots, k\}\)
2: for each \(t_j \in T\) do
3: Compute-Matrix-Column \((H, G, T, t_j)\)
4: initialize \(k\) primary corridors: \(PC = \{pc_j | j = 1, \ldots, k\}\)
5: while \(PC\) is not updated do
6: for each track \(t_i \in T\) do
7: for each \(pc_{0j} \in PC\) do
8: look up Hausdorff distance \(H(t_i, pc_{0j})\)
9: find the \(pc_j\) minimizing \(H(t_i, pc_{0j})\)
10: assign \(t_i\) to the \(pc_j\) (i.e., cluster \(T_j\) \(\rightarrow\) \(T_j\) \(\cup\) \(\{t_i\}\))
11: for each track cluster \(T_j\) do
12: for each candidate track \(pc_{0j} \in T_j\) do
13: sum \(\leftarrow 0\)
14: for each remaining track \(t_j \in T_j\) \(\setminus\) \(\{pc_{0j}\}\) do
15: look up Hausdorff distance \(H(t_j, pc_{0j})\)

Algorithm 3 describes how to precompute a column of the distance matrix (i.e., Hausdorff distances from all tracks to a given track) in one Dijkstra call. The main idea is to initialize the priority queue in Dijkstra in a special manner such that all nodes on the track have zero weight [20]. This special initialization can be done via a new virtual node \(v_0\), which is connected to every node in the track by a zero-weight edge (steps 1–3). A Hausdorff distance from a source track can be computed as the maximum shortest path distance over its nodes (steps 6–8). For example, in Fig. 5(a), with one Dijkstra call from the virtual node on \(t_{10}\), we can get the shortest path distances from all other nodes to \(t_{10}\). After scanning the maximums on different source tracks, their Hausdorff distances to \(t_{10}\) are computed all together (i.e., the last column of the matrix in Fig. 5(b)).

With this column-wise pre-computation of a lookup table, the number of Dijkstra calls is reduced from \(n^2\) in the brute-force approach to \(n\), where \(n\) is the number of tracks. However, one Dijkstra call is still needed on every column of the matrix. These shortest path computations remain as the main bottleneck, especially when the number of tracks and the road network size are large.

![Fig. 5](image-url) An illustrative example of multi-source Dijkstra. (a) a virtual node, (b) column of distances to the target track \(t_{10}\) (best viewed in color). (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)
16: \[ \text{sum} = -\text{sum} + H(t_i, p_{c_0}) \]
17: \[ p_{c_j} = p_{c_0} \text{ with minimum sum} \]
18: \[ \text{return } p_C = \{p_{c_j}, j = 1, \ldots, k\} \]

**Algorithm 3.** Compute-Matrix-Column\((H, G, T, t_j)\).

**Input:**
- \(H\): a \(|T|\) by \(|T|\) matrix
- \(G\): an undirected graph representing a road network
- \(T\): a set of GPS tracks on the road network
- \(t_j\): a track to which (column) the exact Hausdorff distances are computed

**Output:**
- update \(H\) column for \(t_j\) with exact Hausdorff distances
1: add virtual node \(v_0\) to \(G\)
2: for each node \(v_j \in t_1\) do
3: add zero-weight edge between \(v_j\) and \(v_0\) to \(G\)
4: call Dijkstra on \(G\) with source node \(v_0\)
5: for each \(t_j \in T\) do
6: for each node \(v_k \in t_j\) do
7: IF \(d(v_k, v_0) > H(t_i, t_j)\) then
8: \[ H(t_i, t_j) = d(v_k, v_0) \]
9: remove node \(v_k\) and its edges from \(G\)
10: return \(H\)

3.4. New lower bound filtering approach

To further reduce the computational overhead of shortest path distance computation, we refine our lookup table approach and propose a new algorithm with a lower bound filter. The key idea is as follows: first, in the lookup table, we compute the lower bounds of distances (computationally cheaper), and only recompute exact distances when necessary; second, during primary corridor updating, if the lower bound of the total distances of a candidate track is higher than the current minimum, this candidate can be filtered out without computing its exact distances.

3.4.1. Overall algorithm structure

The main structure of our new algorithm is in Algorithm 5. The algorithm is very similar to Algorithm 2, but it has three main differences. First, step 1 precomputes a lookup table (distance matrix) of lower bounds of Hausdorff distances, instead of exact Hausdorff distances, via the Compute-Matrix-Lower-Bound subroutine (Algorithm 6). Step 2 initializes a boolean vector indicating all columns are lower bounds (not exact distances). Second, in the track assignment phase, steps 7–9 recompute exact distances in the column of a primary corridor if the current distances are only lower bounds (i.e., isExact is false). This is necessary since we need to know exactly which primary corridor is the closest. Third, in the primary corridor updating phase, step 14 initializes a priority queue, and step 20 populates the priority queue with candidate tracks ordered by the sums of their distances from remaining tracks. A new primary corridor is selected by the new Update-Primary-Corridor subroutine (Algorithm 7). The details of how to compute lower bounds and how to use a lower bound filter to update primary corridors are introduced separately below.

3.4.2. Track envelope and lower bound of Hausdorff distance

The lower bounds of Hausdorff distance can be computed by first dividing a collection of tracks into small track envelopes, and then computing shortest path distances to these envelopes.

**Track envelope:** The track envelope of a number of tracks is defined as the union of all their nodes. For example, Fig. 6(a) shows four track envelopes surrounded by dashed lines. The track envelope containing \(t_3\) and \(t_4\) has 9 nodes. A track envelope has a lower bound property described in the following lemma.

**Lemma 1.** \(H(t_i, e) \leq H(t_i, t_j), \forall t_j \subseteq e,\) where \(H\) is Hausdorff distance, \(t_i\) and \(t_j\) are tracks, and \(e\) is a track envelope containing \(t_j\). In other words, the Hausdorff distance to a track envelope is a lower bound of the distance to any member track.

**Proof.** According to the definition of Hausdorff distance, \(H(t_i, t_j) = \max_{v_i \in t_i} \min_{v_j \in t_j} d(v_i, v_j)\). Since \(t_j \subseteq e\), we have \(\min_{v_j \in e} d(v_i, v_j) \leq \min_{v_j \in t_j} d(v_i, v_j)\). Then, \(\max_{v_i \in t_i} \min_{v_j \in e} d(v_i, v_j) \leq \max_{v_i \in t_i} \min_{v_j \in t_j} d(v_i, v_j)\). Thus, \(H(t_i, e) \leq H(t_i, t_j)\).

**How to form track envelopes?** There are two main considerations in track envelope formation: the lower bound of Hausdorff distance should be as tight as possible, and envelope formation should be computationally simple. We
new introduce three strategies to form track envelopes with a fixed size and discuss factors influencing the tightness of the lower bounds. Algorithm 4 shows the details of these strategies. The first strategy is to randomly divide the tracks into groups of a fixed size $S_e$. This is the simplest way with a linear cost of $O(|T|)$, where $|T|$ is the total number of tracks. The other two strategies are based on the intuition that tracks that are “close” or “similar” to each other should be grouped into the same envelope. These two strategies use different measures of “closeness” or “similarity”: one uses size of overlap $(|t_i \cap t_j|)$ between tracks and the other uses Jaccard coefficient $(J(t_i, t_j) = \frac{|t_i \cap t_j|}{|t_i| + |t_j| - |t_i \cap t_j|})$ between tracks, considering each track as a set of nodes. More details are given in steps 4–14 of Algorithm 4. Steps 4–8 compute a track similarity matrix. If using bitmaps, these steps have a time complexity of $O(|T|^2 |t|)$ where $|t|$ is the length (number of nodes) of tracks. Steps 9–13 group these tracks using a greedy strategy. A set $R$ of remaining ungrouped tracks is maintained. As long as $R$ is not empty, the algorithm randomly extracts a track $t$ from $R$ and then extract the $S_e - 1$ most similar tracks of $t$ from $R$. These $S_e$ tracks are combined into a new track envelope. This process continues until $R$ is empty, when all tracks have been grouped into envelopes. If finding $S_e - 1$ most similar tracks is done via a binary heap, steps 10–13 have a time complexity of $O(|T|^2 \log |T|) = O(|T|^2 + S_e|T| \log |T|)$, where $S_e$ is the total time complexity of the “overlap” and “Jaccard coefficient” strategies is dominated by $O(|T|^2 |t|)$. Assuming the number of tracks $|T|$ is much smaller than the number of nodes in a road network, the time complexity of forming track envelopes is much lower than Hausdorff distance computation.

Factors influencing the tightness of lower bounds: We now discuss the tightness of the lower bounds of Hausdorff distances $(\text{LowerBound}(H(t_i, t_j)) = H(t_i, e))$, where $t_j \subseteq e$. We consider three factors: the size of envelopes $S_e$, the envelope formation strategy, and the actual distance $H(t_i, t_j)$ (influenced by $k$).

- Envelope size $S_e$: Our intuition is that the smaller the $S_e$, the tighter the lower bound. This can be proved as follows. Assume a source track $t_s$, a target track $t_t$ as well as two envelopes $e_1$ and $e_2$ such that $t_t \subseteq e_1 \subseteq e_2$. (If $e_2$ is larger than $e_1$, we now compare the lower bounds of $H(t_t, e_1)$. The lower bound from $e_1$ is computed as $\text{LowerBound}_{e_1}(H(t_t, e_1)) = H(t_t, e_1)$. The lower bound from $e_2$ is computed as $\text{LowerBound}_{e_2}(H(t_t, e_2)) = H(t_t, e_2)$. It can be proved in a similar way as in Lemma 1 that $H(t_t, e_1) \geq H(t_t, e_2) \geq H(t_t, e_2)$. Thus, the lower bound from the smaller envelope $e_1$ is tighter. However, this conclusion does not mean a smaller $S_e$ is always better, since a smaller $S_e$ means more track envelopes and more lower bound computations. This issue will also be discussed in more detail in the theoretical analysis section.

- Envelope formation strategy: Our second intuition is that the larger the ratio of overlap between tracks within an envelope, the tighter the lower bound tends to be. Without the loss of generalizability, we prove this intuition for envelopes of size two as shown in Fig. 7. Suppose there is no prior information on where $t_i$ is.

Then $H(t_i, e)$ can be found as a shortest path distance to any node in $e$ with equal probability. If it is found on an overlapping node $v \in (t_i \cap t_j)$, then $H(t_i, t_j) = H(t_i, v) = H(t_j, e)$, which means the lower bound $H(t_i, e)$ is tight for both $H(t_i, t_j)$ and $H(t_j, t_i)$. This is the best case and its probability is the Jaccard coefficient $J(t_i, t_j) = \frac{|t_i \cap t_j|}{|t_i| + |t_j| - |t_i \cap t_j|}$. Thus, we expect track envelopes formed by Jaccard coefficients to provide the tightest bound, envelopes formed by overlaps to be a little less tight, and envelopes formed randomly to be the worst, if no other information on $t_i$ is given.

- Actual distance $H(t_i, t_j)$ (influenced by $k$): Here, our intuition is that the further away the source track $t_i$ is from the track envelope, the tighter the lower bound of distance from source track to target tracks within the envelope tends to be. This can be illustrated in Fig. 7(a). If $t_i$ is very far away from $t_j$, then the relative differences between $H(t_i, e)$, $H(t_j, t_i)$, and $H(t_i, t_j)$ are very small. Thus, the lower bound is tight. The actual distance $H(t_i, t_j)$ is influenced by the total number of primary corridors $k$. A smaller $k$ means less “clusters” or partitions of tracks, and thus more tracks in each partition or “cluster” (larger clusters). Since the locations of tracks are fixed, a larger cluster of tracks means higher average distances $H(t_i, t_j)$ between tracks within the cluster. Therefore, a smaller $k$ tends to make the lower bound tighter.

Algorithm 4. FormTrackEnvelope($T, S_e, \text{method}$).

**Input:**
- $T$: a set of GPS tracks on the road network
- $S_e$: size (number of tracks) of trace envelopes
- $\text{method}$: “random”, “overlap”, or “Jaccard”

**Output:**
- $\text{trace envelopes}$
- $T$: a set of GPS tracks on the road network

**Algorithm 4.** FormTrackEnvelope($T, S_e, \text{method}$).

1. **if** method is “random” **then**
2. randomly partition $T$ into groups of size $S_e$
3. **return** groups as track envelopes
4. **initialize** similarity matrix $\text{sim}[[t]]$ (|T| by |T|)
5. **if** method is “overlap” **then**
6. compute $\text{sim}[[t]]$ as overlap between tracks
7. **else if** method is “Jaccard” **then**
8. compute $\text{sim}[[t]]$ as jaccard coefficients
9. **initialize** a set of remaining tracks $R \leftarrow T$
10. **while** $R$ is not empty **do**
11. randomly remove track $t$ from $R$
12. remove $S_e - 1$ (or less if $R$ gets empty) tracks with highest similarities with $t$ in $R$
13. if $\text{sim}[[t]]$ is satisfied **then**
14. **compute** $H(t_i, t_j)$ **end**
15. **end while**
16. **end if**
17. **end if**
18. **end Algorithm 4.**
13: combine track \( t \), the \( S_t - 1 \) tracks as an envelope
14: return all track envelopes

How to compute lower bounds? The Compute-Matrix-Lower-Bound subroutine (Algorithm 6) is very close to Algorithm 3, which computes the exact Hausdorff distances in a matrix column. The difference is that Algorithm 6 divides the set of input tracks into track envelopes (step 1), and calls one Dijkstra on a virtual node to one envelope, instead of a single track (steps 4–6). Thus, the lower bounds of several columns corresponding to member tracks of the envelope can be computed all together (steps 11–12). An illustrative example is in Fig. 6, where the last three columns are computed in one Dijkstra call.

Why are lower bounds computationally cheaper than exact distances? It has been shown above that the cost of track envelope formation is negligible compared with Hausdorff distance computation, and that the Hausdorff distance from a track to a track envelope can be computed in one Dijkstra call. If the envelope size is \( S_t \), then a set of \( S_t \) lower bounds between the source track and target tracks within the envelope is computed in one Dijkstra call. Thus, each lower bound is costing \( 1/S_t \). Dijkstra call on average, while each exact Hausdorff distance is costing one Dijkstra call. The larger the track envelope size \( S_t \) is, the smaller the cost of the lower bound computation.

3.4.3. Lower bound filtering with a priority queue

The goal is to select a candidate track with the minimum exact sum of distances. Candidate tracks are populated into a priority queue ordered by the lower bounds of their sum of distances. In this way, the most promising remaining candidate can be evaluated first. If the lower bound of the most promising candidate is higher than the current minimum exact sum so far, all the remaining candidates in the queue can be filtered out.

The Update-Primary-Corridor subroutine in Algorithm 7 shows more details. Step 1 initializes two variables maintaining the best candidate primary corridor (pc) evaluated so far, as well as its exact sum of distances. Steps 2–15 evaluate candidates from the priority queue. More specifically, step 3 pops a candidate from the queue. If its lower bound is higher than that of the current best, the loop is terminated (step 5). Otherwise, the algorithm checks whether the candidate’s exact sum is lower than the current minimum. Steps 6–12 compute the exact sum of distances if the candidate’s sum is a lower bound (indicated by the boolean variable of isExact for its column). Steps 13–15 update the current best candidate and its exact sum as needed. Step 16 returns the new primary corridor.

An example is given in Fig. 8. The candidate tracks in the priority queue are tracks \( t_3 \) to \( t_{10} \) from Fig. 6(a). The vertical axis shows their exact sum of distances, while the horizontal axis shows their “priority” (ranked by lower bounds of the sum of distances). As can be seen, tracks from the same track envelope have the same lower bounds. Tracks \( \{t_5, t_6, t_7\} \) have the highest priority, and \( t_5 \) is evaluated first. The current minimum exact sum is indicated by the horizontal dash line. When evaluating the next candidate \( t_6 \), the algorithm finds that the lower bound of \( t_6 \) is no lower than the current minimum. Thus, the remaining candidates are all filtered out. Candidate \( t_5 \) is the new primary corridor. Note that this is the most favorable case. Very often, a number of candidates needs to be evaluated in order to ensure that the current minimum exact sum is low enough, the remaining candidates’ lower bounds are high enough, and thus terminate the evaluation process.

Algorithm 5. K-Primary-Corridor-LBFilter \((G, T, k)\).

Input:
- \( G \): an undirected graph representing a road network
- \( T \): a set of GPS tracks on the road network
- \( k \): the number of primary corridors

Output:
- \( G \): a \([|T|, |T|]\) matrix of lower bounds of Hausdorff distances

1: \( TEn, m = 1 \rightarrow FormTrackEnvelopes(T) \)
2: for each track envelope \( TEm \) do
3: add virtual node \( v_{sel} \) to \( G \)
4: for each node \( v_{sel} \in TEm \) do


Input:
- \( G \): an undirected graph representing a road network
- \( T \): a set of GPS tracks on the road network

Output:
- \( G \): a \([|T|, |T|]\) matrix of lower bounds of Hausdorff distances
1: \( TEn, m = 1 \rightarrow FormTrackEnvelopes(T) \)
2: for each track envelope \( TEm \) do
3: add virtual node \( v_{sel} \) to \( G \)
4: for each node \( v_{sel} \in TEm \) do
5: add zero-weight edge between \( v_{n+1} \) and \( v_0 \) to \( G \)
6: call Dijkstra on \( G \) with source node \( v_0 \)
7: for each \( t_i \in T \) do
8: for each node \( v_g \in t_i \) do
9: if \( d(v_g, v_0) > D(t_i, T_{Em}) \) then
10: \( H(t_i, T_{Em}) = d(v_g, v_0) \)
11: for each track \( t_j \in T_{Em} \) do
12: \( H(t_i, t_j) = H(t_i, T_{Em}) \)
13: remove node \( v_g \) and its edges from \( G \)
14: return \( H \)

Algorithm 7. Update-Primary-Corridor (\( H, \) isLowerBound, \( G, T, PQ \)).

Input:
- \( H \): a \( |T| \times |T| \) matrix of Hausdorff distances
- isExact: a vector indicating if an \( H \) column is exact distance
- \( G \): an undirected graph representing a road network
- \( T \): a set of GPS tracks on the road network
- PQ: a priority queue of tracks by sum of distances

Output:
- PC: a new primary corridor from candidates in PQ

1: initialize \( \text{cur}_\text{min}_\text{sum} = +\infty, \text{cur}_\text{best}_\text{pc} = \emptyset \)
2: while PQ not empty do
3: \( (\text{pc}_\text{cur}_\text{pc}, \text{sum}) = \text{PQ}.\text{pop}() \)
4: if \( \text{sum} \geq \text{cur}_\text{min}_\text{sum} \) then
5: terminate the while loop
6: if isExact(\text{pc}_\text{cur}_\text{pc}) is false then
7: Compute-Matrix-Column (\( H, G, T, \text{cur}_\text{pc} \))
8: isExact(\text{pc}_\text{cur}_\text{pc}) = true
9: \( \text{sum} = 0 \)
10: for each remaining track \( t_i \in T \setminus (\text{pc}_\text{cur}_\text{pc}) \) do
11: look up Hausdorff distance \( H(t_i, \text{pc}_\text{cur}_\text{pc}) \)
12: \( \text{sum} = \text{sum} + H(t_i, \text{pc}_\text{cur}_\text{pc}) \)
13: if \( \text{sum} < \text{cur}_\text{min}_\text{sum} \) then
14: \( \text{cur}_\text{min}_\text{sum} = \text{sum} \)
15: \( \text{cur}_\text{best}_\text{pc} = \text{pc}_\text{cur}_\text{pc} \)
16: return \( \text{cur}_\text{best}_\text{pc} \)

3.4.4. A running example

We now illustrate Algorithm 5 with a running example in Fig. 9. The inputs are the same as those used in Figs. 3 and 4. There are ten tracks (\( t_1 \) to \( t_{10} \), \( k = 2 \), and the initial primary corridors are \( t_1 \) and \( t_3 \).

The pre-computation of matrix \( H \) of distance lower bounds (steps 1 and 2) is shown in Fig. 6.

Track assignment (steps 5–12) in the first iteration recomputes exact distances for the two columns of primary corridors \( t_1 \) and \( t_3 \), and then assigns all tracks to their closest primary corridors (Fig. 9(a)).

Primary corridor updating (steps 13–21) pushes candidates from each cluster (i.e., tracks assigned to the same primary corridor) into a priority queue, ranked by the lower bounds of the sum of the distances. The two tables of Fig. 9(b) represent two priority queues. The Update-Primary-Corridor subroutine (Algorithm 7) is then used to select a new primary corridor. We further explain this subroutine with the second priority queue Fig. 9(b) (i.e., the table with tracks \( \{t_5, t_6, t_7, t_8, t_9, t_{10}, t_4, t_3\} \)). After initialization, the subroutine keeps "popping" candidates and evaluating them. Track \( t_5 \) is the first candidate popped from the queue, and its lower bound of sum of distances is 12, lower than the current minimum \( +\infty \). Thus, the subroutine does not terminate. It recomputes the exact distances in the \( t_5 \) column and calculating its exact sum which is 12. It updates the current minimum sum from \( +\infty \) to 12. The next candidate \( t_6 \) is popped from the queue, and the lower bound of its sum of distances is 12, no smaller than the current minimum. Thus, the subroutine terminates the loop and returns \( t_5 \) as the new primary corridor, without recomputing the exact distances for the remaining candidates. The new primary corridor for the first cluster is still \( t_1 \).

The second iteration is similar. The track assignments (Fig. 9(c)) are updated as \( \{t_1, \ldots, t_4\} \) and \( \{t_5, \ldots, t_{10}\} \). The new primary corridors are still \( t_1 \) and \( t_5 \) as shown in Fig. 9 (d). Finally, the algorithm terminates with final primary corridors as \( t_1 \) and \( t_5 \).

4. Theoretical analysis

We provide theoretical analysis on the three computational algorithms introduced in Section 3. We prove that our new algorithm with a lower bound filter (Algorithm 5) is correct. We then analyze the computational cost models of the algorithms.

4.1. The correctness of the lower bound filter

Theorem 1. Algorithm 5 is correct, i.e., it returns the same output as Algorithm 2.

Proof. Algorithm 5 consists of three parts: pre-computation (steps 1 and 2), track assignment (steps 7–9), and primary corridor updating (step 14 and step 21).

In the pre-computation phase, the computed matrix \( H \) does contain lower bounds of Hausdorff distances (already proved by Lemma 1 in Section 3.4.2).

During the track assignment, steps 7–9 re-compute the exact distances to primary corridors if current distances are only lower bounds (i.e., isExact(\text{pc}_\text{cur}_\text{pc}) is false). The track assignment still uses exact Hausdorff distances and thus is correct.

In the primary-corridor-updating phase, the algorithm populates a minimum priority queue with candidate tracks for each cluster (steps 14–20). The weight or priority is based on the lower bounds of sum of distances. Then the Update-Primary-Corridor subroutine (Algorithm 7) is called to return the new primary corridor for the cluster. This subroutine can be proved to return a new primary corridor with the minimum sum of distances (SOD) as follows. The subroutine maintains the current best candidate based on the exact SOD, and compares it with a candidate generated from the priority queue (i.e., candidate with minimum SOD lower bound). If the current best candidate’s exact SOD is at all lower than the minimum lower bound, then the candidate is returned as the new primary corridor. Otherwise, the exact SOD of the candidate is computed and the current best candidate is updated if necessary. Thus, the primary-corridor-updating phase is also correct. □
4.2. Computational cost models

We now analyze the computational complexity of the algorithms discussed in Section 3. The variables used in our analysis are defined in Table 1. All variables can be determined by the input data except $\alpha$ and $I$. The pruning ratio $\alpha$ reflects the tightness of lower bounds, which is determined by track envelope size $S_e$, the envelope

![Fig. 9. A running example of the K-Primary-Corridor algorithm with a lower bound filter.](image-url)
In each iteration, the track assignment phase computes the minimum in the Hausdorff distance computation. The brute-force approach, as well as choice of initial primary corridors, as well as as well as choice of initial primary corridors.

4.2.1. Brute-force approach

**Theorem 2.** The brute-force approach (Algorithm 1) has $O(I \cdot |t| \cdot |T|^2)$ Dijkstra calls. If Dijkstra is implemented by a min-heap, the total cost is $O(I \cdot |t| \cdot |T|^2 \cdot (|V| \log |V| + |E| \log |V|))$.

**Proof.** In each iteration, the track assignment phase computes $k \cdot |T|$ Hausdorff distances, and the primary corridor updating phase computes $O(|T|^2)$ Hausdorff distances. Thus, the entire algorithm computes $O(I \cdot |T|^2)$ Hausdorff distances in total. Since it computes Hausdorff distances on the fly, each Hausdorff distance needs $|t|$ Dijkstra calls. Thus, the total number of Dijkstra calls is $O(I \cdot |T|^2 \cdot |t|)$. A min-heap implementation of Dijkstra has a cost of $O(|V| \log |V| + |E| \log |V|)$. Thus, the total cost of the brute-force approach is $O(I \cdot |t| \cdot |T|^2 \cdot (|V| \log |V| + |E| \log |V|))$. □

4.2.2. Column-wise lookup table approach

**Theorem 3.** The column-wise lookup table approach (Algorithm 1) needs $|T|$ Dijkstra calls if Dijkstra is implemented with a min-heap, the total cost is $O(|T| \cdot (|V| \log |V| + |E| \log |V| + |T| \cdot |t| + I \cdot |T|^2))$.

**Proof.** The pre-computation of a lookup table involves $O(|T|)$ Dijkstra calls as well as a cost of $O(|T|^2 |t|)$ to scan the minimum in the Hausdorff distance computation. The iteration part has a cost of $O(|T| \cdot |T|^2)$. Thus, the total cost of min-heap implementation is $O(|T| \cdot (|V| \log |V| + |E| \log |V| + |T| \cdot |t| + I \cdot |T|^2))$. □

4.2.3. Lower bound filtering approach

**Theorem 4.** The lower bound filtering approach (Algorithm 5) needs $\frac{1}{\alpha} \cdot (1 - \alpha) \cdot |T|$ Dijkstra calls. If Dijkstra is implemented by a min-heap, the total cost is $O\left(\left(\frac{1}{\alpha} + 1 - \alpha\right) \cdot |T| \cdot (|V| \log |V| + |E| \log |V| + |T| \cdot |t| + I \cdot |T|^2), plus a cost of \right) O(|T|)$ for random envelope creation or $O(|T|^2 |t|)$ for “overlap” or “Jaccard coefficient” based envelope creation.

**Proof.** Given $|T|$ tracks and an envelope size $S_k$, the number of track envelopes is $\frac{1}{\alpha}$, The costs of forming track envelopes with three different strategies are already analyzed in Section 3.4.2. Additionally, each track envelope requires one call of Dijkstra, and thus there are $\frac{1}{\alpha}$ calls of Dijkstra in total. The iteration part has $(1 - \alpha) \cdot |T|$ Dijkstra calls, assuming $|t| \cdot |T|$ calls are pruned out. For each Hausdorff distance computation, a cost of $O(|T|^2 |t|)$ is needed to scan all nodes in the tracks after a Dijkstra call. The iteration process has a remaining cost of $O(I \cdot |T|^2)$. Thus, $\frac{1}{\alpha} \cdot (1 - \alpha) \cdot |T|$ Dijkstra calls. If Dijkstra is implemented by a min-heap, the total cost is $O\left(\left(\frac{1}{\alpha} + 1 - \alpha\right) \cdot |T| \cdot (|V| \log |V| + |E| \log |V| + |T| \cdot |t| + I \cdot |T|^2)\right)$. □

Factors controlling $\alpha$: The pruning ratio $\alpha$ is an important factor in the computational complexity of our lower bound filtering approach. It depends on the tightness of the lower bounds computed from the track envelopes. According to our discussion in Section 3.4.2, in order to have a high pruning ratio $\alpha$ (or tighter bounds), we need a smaller $S_k$ and $k$, as well as a good track envelope formation strategy. The “Jaccard coefficient” is the most preferable, “overlap” is a little less preferable, and “random” is the least preferable.

4.2.4. Comparison of cost models of the three approaches

It is obvious that the brute-force approach has much higher time complexity than the other two approaches. We now compare our previous column-wise lookup table approach and the lower bound filtering approach. The main differences in their cost models are that the former has $|T|$ Dijkstra calls while the latter has $\left(\frac{1}{\alpha} + 1 - \alpha\right) \cdot |T|$ Dijkstra calls $\left(\frac{1}{\alpha} + 1 - \alpha\right) \cdot |T|$ may be smaller or larger than $|T|$ depending on $\alpha$, and that the latter has a small additional track envelope formation cost (i.e., $|T|$ for random strategy and $|T|^2 |t|$ for “overlap” or “Jaccard coefficient” strategy). If we assume the road network is large and $|V| \log |V| + |E| \log |V| + |T| \cdot |t|$, then the track envelope formation cost can be ignored and the number of Dijkstra calls is the dominating factor.

The number of Dijkstra calls for each approach is summarized in Table 2. As can be seen, the brute-force approach makes a large number $(k \cdot |T|^2 \cdot |t|)$ of Dijkstra calls. Our previous column-wise lookup table approach reduces the number of Dijkstra calls to $|T|$. Our new lower bound filtering approach has $\left(\frac{1}{\alpha} + 1 - \alpha\right) \cdot |T|$ calls. Our lower bound filtering approach is better than our previous column-wise lookup table approach if $\frac{1}{\alpha} + 1 - \alpha < 1$ (i.e., $\alpha > \frac{1}{\alpha}$), and is worse otherwise. In practice, in order to make $\alpha > \frac{1}{\alpha}$, we need to select a preferable $S_k$ and track envelope formation strategy. According to our discussion on factors controlling $\alpha$ in Section 4.2.3, in order to have high pruning ratio $\alpha$, we should prefer a smaller $k$ and $S_k$, and track envelope formation based on "Jaccard coefficient", or "overlap" rather than "random". However, although a smaller $S_k$ contributes to higher $\alpha$, it also leads to higher $\frac{1}{\alpha}$ (i.e., more envelopes). Thus, we would prefer a

| Table 1 |
| List of symbols and descriptions. |
| Symbol | Description |
| $|t|$ | A track, and track length (number of nodes) |
| $|T|$ | The set of all input tracks, and size of the set |
| $I$ | Number of iterations in an execution |
| $k$ | Number of primary corridors |
| $S_k$ | Size of track envelopes in lower bound filtering |
| $\alpha$ | Pruning ratio, $0 \leq \alpha < 1$ |
| $|V|$ | Number of nodes in a road network |
| $|E|$ | Number of edges in a road network |

| Table 2 |
| Comparison of shortest path computation cost. |
| Approach | Number of Dijkstra calls |
| Brute-force | $k \cdot |T|^2 \cdot |t|$ |
| Column-wise lookup table | $|T|$ |
| Lower bound filtering | $\left(\frac{1}{\alpha} + 1 - \alpha\right) \cdot |T|$ |
5. Evaluation

The goal of our experiments and case study was to investigate the following questions:

- Does our new algorithm with a lower bound filter reduce the computational cost of our previous algorithm with column-wise lookup table only?
- How sensitive is the computational performance of lower bound filtering to the parameters of \( k \), \( S_e \) as well as the track formation strategy?
- How does our approach compare with density-based hot-route detection in identifying primary corridors in a real world case study?

5.1. Experiment setup

**Experiment design**: The experiment design is shown in Fig. 10. The inputs were a road network, a set of GPS tracks on the network, as well as configuration parameters such as number of output primary corridors \( k \), track envelope size \( S_e \) and envelope formation strategy. To evaluate computational performance, we compared the column-wise lookup table approach proposed in our Urban Computing 2013 paper [8] (now called the “table lookup approach”) and the new lower bound filtering approach proposed here on a real world urban bicyclist GPS trajectory dataset. A comparison of the brute force approach and the table lookup approach was done previously in [8] and thus is not included here. Computational time reported was the average of 100 runs. To evaluate the effectiveness of our approach, we ran a case study to compare corridors output from our approach and from a density-based hot route detection method on a real world dataset of cyclists’ trajectories in Minneapolis, MN. All the algorithms were implemented in Java language. Experiments were conducted on a Dell workstation with Intel(R) Xeon(R) CPU E5-1607 at 3 GHz, and 64 GB RAM.

*The real world dataset* was collected in 2006 by our colleagues from the Department of Geography of the University of Minnesota [10]. The goal was to gain a better understanding of commuter bicyclist behavior using GPS equipment and personal surveys to record bicyclist movements and behaviors. The dataset consists of 819 trips (i.e., GPS tracks) between commuter cyclists’ homes in south Minneapolis and their work locations near downtown Minneapolis (shown in Fig. 15(a)). GPS points were map matched to a road network from Tiger Shapefile [25] with 14,664 nodes and 23,564 edges covering the study area.

5.2. Computational performance evaluation

This section compares the computational performance of our two algorithms on the real world dataset. Though the real world dataset is of small size (with 819 tracks), it reflects the distribution of real world GPS tracks on a road network. Symbols are explained in Table 1.

5.2.1. Effect of number of tracks

We fixed the number of primary corridors to \( k = 10 \), and the size of track envelopes to \( S_e = 5 \). We created six sets of tracks of different sizes by slicing the tracks temporally. The number of tracks \(| T |\) are 136, 256, 369, 449, 569, and 819 respectively.

The computational time costs of Lower Bound Filtering with different filter strategies and our previous Table Lookup approach are shown in Fig. 11. As can be seen, the time costs of all approaches increase almost linearly with the number of tracks. This confirms our theoretical analysis that the dominating factor (i.e., shortest path computation) in the cost models is linear with \(| T |\). Lower Bound Filtering with “random” envelopes costs almost the same as (slightly worse than) Table Lookup alone. Filtering with “overlap” or “Jaccard” envelope strategies reduces the cost of Table Lookup alone approach by around one third
to one half, with the “Jaccard” strategy performing slightly better. More specifically, as the number of tracks increases, the pruning ratio gets higher. The reason is that more overlapping tracks are added as the number of tracks gets larger, making the envelopes tighter.

The number of Dijkstra calls required for different approaches is summarized in Table 3. The “filter”, “refine”, and “total” columns in the table correspond to the filtering (i.e., track envelope formation), refinement, and total phases respectively in Lower Bound Filtering; the last column corresponds to our previous Table Lookup approach without filtering. As can be seen, the number of Dijkstra calls in the filtering phase for “random” envelopes, “overlap” based envelopes, and “Jaccard coefficient” based envelopes is the same, and equals $|T|$.

The refinement phase for “overlap” and “Jaccard coefficient” based envelopes has a much smaller number of Dijkstra calls than refining for random envelopes, confirming the effectiveness of these pruning strategies. The total number of Dijkstra calls also shows the same trend as Fig. 11.

### 5.2.2. Effect of number of corridors $k$

We fixed the number of tracks as $|T| = 819$, and the size of track envelopes as $S_e = 5$. We varied the number of primary corridors $k$ in the algorithms from 2 to 10 and then from 20 to 100. In practice, given 819 tracks and the small study area of south Minneapolis, the number of primary corridors should be small (e.g., less than 10). Otherwise, there will be too few tracks within the track cluster for a given primary corridor.

Fig. 12 shows the results of these tests. As can be seen, the cost of Table Lookup alone remains almost unchanged when $k$ increases. The reason is that the dominating factor of computational cost, i.e., the number of Dijkstra calls, is fixed as $|T|$. The cost of the lower bound filtering approach shows different trends for different track envelope strategies. For “overlap” and “Jaccard coefficient” based envelopes, the computational cost increases with $k$ but always stays lower than the previous table lookup alone approach. This trend confirms our theoretical analysis in Sections 4.2.3 and 4.2.4 that a smaller $k$ contributes to longer distances between source tracks and target envelopes, and thus tighter lower bounds and a higher pruning ratio $\alpha$. The “Jaccard coefficient” based envelopes perform slightly but persistently faster than the “overlap” based envelopes, which is consistent with our theoretical analysis that the bounds from “Jaccard coefficient” are tighter. What is interesting in the figure is the behavior of random envelopes for different $k$. As $k$ increases, the cost of lower bound filtering with random envelopes continuously decreases, and becomes lower than the cost of table lookup alone approach after $k$ reaches around 8. The reason is illustrated in Fig. 13. When $k$ is very large, there will be very few tracks within each track cluster, and thus a random envelope probably consists of tracks from different track clusters as shown by the dashed line. In this case, when computing Hausdorff distance from one track to another track within the same cluster, the lower bound of this distance should be very close to the actual distance.

![Fig. 12. Computational performance comparison with different number of primary corridors.](image-url)

![Fig. 13. Illustration on increasingly tight lower bounds from random envelopes for large $k$. Small circles represent tracks, larger circles represent track clusters, the dash line connects tracks in an envelope.](image-url)
value of $k$ is often small in practice, and the randomly formed envelopes are much less preferable due to their poor performance for a small $k$. This is already shown in the results of the first experiment in Fig. 11.

The number of Dijkstra calls in different approaches for different $k$, shown in Table 4, is consistent with our analysis above. Dijkstra calls in the refinement phase number the same for all three strategies, i.e., 164 = $\left\lceil \frac{156}{5} \right\rceil$. Dijkstra calls for lower bound filtering with “overlap” based and “Jaccard coefficient” based envelopes increase with $k$ but stay lower than the number in our previous lookup table approach. In contrast, the total number of Dijkstra calls for lower bound filtering with random envelopes decreases with $k$, dominating the table lookup alone approach after around $k \geq 10$ and dominating the other two envelope strategies after around $k \geq 70$.

### 5.2.3. Effect of track envelope sizes

We fixed the number of tracks $|T| = 819$, and the number of primary corridors $k = 10$. We varied the size of track envelopes $S_e$ from 2, 3, to 10, and then from 20, 30, to 100. The range of values of $S_e$ was selected to test the sensitivity of different lower bound filtering approaches to $S_e$.

The results are shown in Fig. 14. The cost of our previous table lookup approach stays constant since it does not make use of track envelopes. The cost of lower bound filtering with different envelope formation strategies shows different trends: with $S_e$ increasing, the cost for random envelopes first increases, then decreases, while the cost for “overlap” or “Jaccard coefficient” based envelopes first decreases, then increases (“Jaccard coefficient” strategy is slightly better than “overlap” strategy, but both better than random envelopes or the lookup table approach). The cost of different envelope strategies finally converges to a level close to the lookup table approach when $S_e$ is too large. The cost trend for “Jaccard coefficient” or “overlap” based envelopes can be explained by our previous theoretical analysis in Section 4.2.4. that with increasing $S_e$, the number of track envelopes (filtering cost) decreases but the tightness of lower bounds gets weaker and the pruning ratio $\alpha$ also gets poorer. From the experiment results, we can see that the best track envelope size $S_e$ for “Jaccard” and “overlap” based envelopes is around 3 for the 819 tracks, and that as long as $S_e$ is reasonably small (e.g., less than 10), their cost is much cheaper than for our previous table lookup approach. The trend for random envelopes is quite interesting. For small $S_e$ (around 3–5), its cost is reasonably good, higher than for the other two envelope strategies but far less than for table lookup alone approach. As $S_e$ increases further, the cost first increases to be above the lookup table alone approach and then falls to a little bit below it. The reason is that at first, a larger $S_e$ leads to less tight lower bounds, but as $S_e$ becomes large enough, far fewer envelopes are created (reducing the cost in filtering phase), and the total cost decreases.

### Table 4

Number of Dijkstra calls in our approaches for different $k$ ($|T| = 819, S_e = 5$, average in 100 runs).

| $k$ | Filter: “random” | | Filter: “overlap” | | Filter: “Jaccard coefficient” | | No filter |
|-----|------------------|----------------|------------------|------------------|------------------|
|     | Filter | Refine | Total | Filter | Refine | Total | Filter | Refine | Total |
| 2   | 164   | 803   | 967   | 164   | 102   | 266   | 164   | 92    | 257   | 819   |
| 3   | 164   | 764   | 928   | 164   | 139   | 303   | 164   | 118   | 282   | 819   |
| 4   | 164   | 732   | 896   | 164   | 160   | 324   | 164   | 148   | 312   | 819   |
| 5   | 164   | 712   | 876   | 164   | 178   | 342   | 164   | 159   | 323   | 819   |
| 6   | 164   | 701   | 865   | 164   | 205   | 369   | 164   | 179   | 343   | 819   |
| 7   | 164   | 676   | 840   | 164   | 224   | 388   | 164   | 197   | 361   | 819   |
| 8   | 164   | 667   | 831   | 164   | 240   | 404   | 164   | 208   | 372   | 819   |
| 9   | 164   | 656   | 820   | 164   | 259   | 423   | 164   | 223   | 387   | 819   |
| 10  | 164   | 644   | 808   | 164   | 267   | 431   | 164   | 235   | 399   | 819   |
| 20  | 164   | 566   | 730   | 164   | 336   | 500   | 164   | 309   | 473   | 819   |
| 30  | 164   | 517   | 681   | 164   | 373   | 537   | 164   | 349   | 513   | 819   |
| 40  | 164   | 492   | 656   | 164   | 405   | 569   | 164   | 388   | 552   | 819   |
| 50  | 164   | 473   | 637   | 164   | 432   | 596   | 164   | 416   | 580   | 819   |
| 60  | 164   | 458   | 622   | 164   | 460   | 624   | 164   | 440   | 604   | 819   |
| 70  | 164   | 450   | 614   | 164   | 484   | 648   | 164   | 467   | 631   | 819   |
| 80  | 164   | 437   | 601   | 164   | 509   | 673   | 164   | 488   | 652   | 819   |
| 90  | 164   | 433   | 597   | 164   | 517   | 681   | 164   | 506   | 670   | 819   |
| 100 | 164   | 430   | 594   | 164   | 533   | 697   | 164   | 525   | 689   | 819   |

Fig. 14. Computational performance comparison with different track envelope sizes.
The Dijkstra call numbers for the different approaches with varying $S_e$ are shown in Table 5. The trend is consistent with Fig. 14. The number of Dijkstra calls for table lookup approach is a constant, equal to the total number of tracks. The total number of Dijkstra calls for lower bound filtering with random envelopes first increases and then decreases with $S_e$, while the total number for “overlap” or “Jaccard” based envelopes first decreases and then increases with $S_e$. The number of Dijkstra calls in the filtering phase for the three envelope strategies is the same, equal to the number of envelopes \( \frac{1}{3} \). One small surprise is that the converged cost for lower bound filtering is slightly lower than the cost of table lookup alone approach measured by number of Dijkstra calls in Table 5. The reason may be that the number of Dijkstra calls is so close that some other constant factors in the cost models are making a difference.

### 5.3. Effectiveness evaluation: a case study

**Goal of the case study:** The goal of the case study was to evaluate the effectiveness of our K-Primary-Corridor approach by comparing it with existing hot route detection approaches \([17,14,4,15]\) as well as handcrafted routes by geographers. The real world dataset we used is already described in Section 5.1. Since we do not distinguish track directions (i.e., whether tracks go from home to work or from work to home), we could not apply the hot route detection methods. Instead, we implemented a simple hot route detection method similar to the method in \([14]\). First, we counted the number of tracks traversing each road segment and identified “hot” road segments whose counts of tracks were above a given threshold $\epsilon=40$. Then we connected “hot” road segments that were within a minimum network distance $\text{min\_dist} = 100$ m. Each connected group of “hot” road segments was considered as a hot route.

### Choice of initial $k$ primary corridors

Just as the choice of initial centroids may lead to the local optima issue for K-Means clustering \([13]\), the choice of initial $k$ primary corridors may influence the final output of our K-Primary-Corridor algorithms. One strategy is to generate different sets of $k$ random initial primary corridors, run the K-Primary-Corridor algorithms, and select the results that have the minimum overall sum of Hausdorff distances from tracks to their closest primary corridors. Another strategy is to look at the map and select $k$ initial corridors relatively well spread across the entire area. In our case study, we used the first strategy and selected the best results for 10 runs. We acknowledge that selecting the initial $k$ primary corridors is a challenging task and more research is needed in future work.

### Analysis of results

The real world GPS tracks used in the case study are shown in Fig. 15(a), where the small green triangles represent home locations in south Minneapolis, and the small red squares represent work locations. Geographers at the University of Minnesota handcrafted eight primary corridors shown in Fig. 15(b). The eight primary corridors ($k=8$) from our K-Primary-Corridor algorithms are shown in Fig. 15(c). As can be seen, these corridors are evenly distributed across the study area. In addition, well-known bicycle routes were identified in our approach such as the West River Trail and Minnehaha Ave, which also in the handcrafted primary corridors. Thus, our algorithms can help automatically suggest several primary corridors and reduce the time and effort in manually handcrafting corridors. The output hot routes from the density-based approach are shown in Fig. 15(d). Some popular routes were also identified such as part of the West River Trail, Minnehaha Ave, as well as Franklin Ave (a horizontal route). But there are discontinuities along routes as well as small “noisy” segments in the downtown area (at the top of the map) due to imbalanced density distributions for different regions. Through the comparison, we found out that the density-based approach can detect routes with

<table>
<thead>
<tr>
<th>Se</th>
<th>Filter: “random”</th>
<th>Filter: “overlap”</th>
<th>Filter: “Jaccard coefficient”</th>
<th>No filter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Filter</td>
<td>Refine</td>
<td>Total</td>
<td>Filter</td>
</tr>
<tr>
<td>2</td>
<td>410</td>
<td>224</td>
<td>634</td>
<td>410</td>
</tr>
<tr>
<td>3</td>
<td>273</td>
<td>419</td>
<td>692</td>
<td>273</td>
</tr>
<tr>
<td>4</td>
<td>205</td>
<td>552</td>
<td>757</td>
<td>205</td>
</tr>
<tr>
<td>5</td>
<td>164</td>
<td>643</td>
<td>807</td>
<td>164</td>
</tr>
<tr>
<td>6</td>
<td>137</td>
<td>705</td>
<td>842</td>
<td>137</td>
</tr>
<tr>
<td>7</td>
<td>117</td>
<td>737</td>
<td>854</td>
<td>117</td>
</tr>
<tr>
<td>8</td>
<td>103</td>
<td>760</td>
<td>863</td>
<td>103</td>
</tr>
<tr>
<td>9</td>
<td>91</td>
<td>775</td>
<td>866</td>
<td>91</td>
</tr>
<tr>
<td>10</td>
<td>82</td>
<td>788</td>
<td>870</td>
<td>82</td>
</tr>
<tr>
<td>20</td>
<td>41</td>
<td>816</td>
<td>857</td>
<td>41</td>
</tr>
<tr>
<td>30</td>
<td>28</td>
<td>818</td>
<td>846</td>
<td>28</td>
</tr>
<tr>
<td>40</td>
<td>21</td>
<td>818</td>
<td>839</td>
<td>21</td>
</tr>
<tr>
<td>50</td>
<td>17</td>
<td>819</td>
<td>836</td>
<td>17</td>
</tr>
<tr>
<td>60</td>
<td>14</td>
<td>819</td>
<td>833</td>
<td>14</td>
</tr>
<tr>
<td>70</td>
<td>12</td>
<td>819</td>
<td>831</td>
<td>12</td>
</tr>
<tr>
<td>80</td>
<td>11</td>
<td>819</td>
<td>830</td>
<td>11</td>
</tr>
<tr>
<td>90</td>
<td>10</td>
<td>818</td>
<td>828</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>9</td>
<td>819</td>
<td>828</td>
<td>9</td>
</tr>
</tbody>
</table>
more flexible directions (e.g., Franklin Ave), but it may produce noisy or discontiguous routes. In contrast, our K-Primary-Corridor algorithms could detect contiguous routes that were more evenly distributed from different source locations to target locations.

6. Discussions

The K-Primary-Corridor problem investigated here aims to find \( k \) routes that minimize total cost of travel shifting from other routes. Our approach resembles the k-medoid clustering algorithm. The choice of initial \( k \) primary corridors is important, and there may also be an issue of a “local minima”. One way to address that is to try different random initial seeds, and use the ones that minimize the final total distances. Our approach may also be sensitive to outliers, i.e., a few GPS tracks that are far from other tracks. A preprocessing step to remove those tracks may help. The computational algorithms we propose are based on Dijkstra shortest path computation, but other shortest path computation methods can also be used. Finally, there is other work in trajectory mining that utilize lower bound filtering such as [24]. However, their queries and lower bound computation differ significantly from our approach.

7. Conclusions and future work

The paper investigates the K-Primary-Corridor (KPC) problem. The problem is important to a variety of domains, such as transportation services interested in finding primary corridors for public transportation or greener travel (e.g., bicycling) by leveraging emerging GPS trajectory datasets. However, the KPC problem is challenging due to the large number of shortest path distance computations across tracks. Related trajectory mining approaches, e.g., density or frequency based hot-routes, focus on anomaly detection rather than identifying representative corridors.

Fig. 15. Case study input and output. (a) Recorded GPS points from bicyclists in Minneapolis, MN. (b) Original, hand-crafted primary corridors identified by fellow geographers in [10]. (c) 8-Primary Corridors chosen by our algorithm from input GPS tracks. (d) Hot routes detected by the density-based approach.
that minimize overall travel distances from other tracks, and thus may not be effective for the KPC problem. Our recent work focuses on identifying k-primary corridors and proposes a computational algorithm that pre-computes a column-wise lookup table. This paper proposes a new computational algorithm with a lower bound filter based on the concept of track envelopes. We present three track envelope formation strategies and analyze factors influencing the tightness of lower bounds. Theoretical analysis shows that the new algorithm is correct, and more efficient than our previous algorithm. Experimental evaluations and case studies confirm that our algorithms are both effective and efficient.

In the future, we will investigate the choice of initial primary corridors. We may also generalize the technique to spatio-temporal road networks.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant nos. 1029711, IIS-1320580, 0940818 and IIS-1218168 as well as USDOD under Grant no. HM1582-08-1-0017 and HM0210-13-1-0005. We would like to thank Professor Harvey from the department of Geography, Environment and Social Science of our university for providing datasets and helpful comments in our case studies. We would also like to thank Kim Koffolt for improving the readability of the paper.

References