Efficient Non-Learning Similar Subtrajectory Search

Jiabao Jin
jiabaojin@stu.ecnu.edu.cn
East China Normal University
Shanghai, China

Peng Cheng
pcheng@sei.ecnu.edu.cn
East China Normal University
Shanghai, China

Lei Chen
Hong Kong University of Science and Technology
Hong Kong SAR, China
leichen@cse.ust.hk

Xuemin Lin
Shanghai Jiaotong University
Shanghai, China
xuemin.lin@gmail.com

Wenjie Zhang
University of New South Wales
Sydney, Australia
wenjie.zhang@unsw.edu.au

ABSTRACT

Similar subtrajectory search is a finer-grained operator that can better capture the similarities between one query trajectory and a portion of a data trajectory than the traditional similar trajectory search, which requires the two checked trajectories are similar to each other in whole. Many real applications (e.g., trajectory clustering and trajectory join) utilize similar subtrajectory search as a basic operator. It is considered that the time complexity is $O(mn^2)$ for exact algorithms to solve the similar subtrajectory search problem under most trajectory distance functions in the existing studies, where m is the length of the query trajectory and n is the length of the data trajectory. In this paper, to the best of our knowledge, we are the first to propose an exact algorithm to solve the similar subtrajectory search problem in $O(mn)$ time for most of widely used trajectory distance functions (e.g., WED, DTW, ERP, EDR and Frechet distance). Through extensive experiments on three real datasets, we demonstrate the efficiency and effectiveness of our proposed algorithms.

PVLDB Reference Format:
Jiabao Jin, Peng Cheng, Lei Chen, Xuemin Lin, and Wenjie Zhang. Efficient Non-Learning Similar Subtrajectory Search. PVLDB, 16(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The increasing popularity of mobile devices flourishes the generation of trajectory data, which is widely used in many fields (e.g., traffic flow prediction [10, 11], route planning [23]). With the focus of researchers on trajectory data, more and more methods are proposed for analyzing and processing trajectory data.

A significant problem in analyzing trajectory data is to query the most similar trajectory to a given trajectory among the vast amount of trajectories in the database [5, 6, 15, 19, 29, 30]. In real scenarios, it is hard to guarantee that the lengths of two trajectories are same or close to each other. Thus, similar subtrajectory search attracts much attention recently as a more practical method [2, 4, 14, 21, 26],

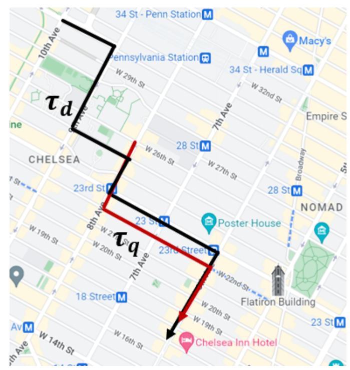


Figure 1: Subtrajectory Search

which uses a part of a long data trajectory as the basic unit to test its similarity to the short query trajectory. For example, as shown in Figure 1, there are two trajectories: data trajectory τ_d and query trajectory τ_q. They are not similar when the whole trajectories are considered, while τ_q is similar to a portion of τ_d.

Searching similar subtrajectories is usually a basic operator in real applications (e.g., subtrajectory join [21] and subtrajectory clustering [2, 4]) and will be frequently invoked, thus its efficiency is very important. One application scenario of subtrajectory query is to analyze the performance of players by their trajectory data in a sport (e.g., soccer or basketball) [26].

Subtrajectory search is a highly related but different problem from trajectory search [8, 9, 13, 20, 27]. Compared with trajectory search, subtrajectory search has to not only consider the data trajectory itself but also determine whether there are subtrajectories of the data trajectory with a smaller distance from the query trajectory. The state-of-the-art study on similar subtrajectory search utilize reinforcement learning methods to accelerate the detecting speed and achieve the time complexity of $O(mn)$ [27], where m is the length of the query trajectory and n is the length of the data trajectory. However, the reinforcement learning based algorithms are approximation algorithms, which have no theoretical guarantee on the accuracy of the returned results. In this paper, we find that *the similar subtrajectory search problem can be solved exactly with the time complexity of $O(mn)$ for most trajectory distance functions* (e.g., DTW, WED, ERP, EDR and FD. Details will be discussed in Section 6), which had not been discovered to the best of our knowledge.

Challenges. For a data trajectory, the number of its subtrajectories is quadratic to its length. Let n be the length of a data trajectory,

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

there will be $\frac{n(n+1)}{2}$ its subtrajectories. Assuming that the length of the query trajectory is m and the length of the data trajectory is n, the time complexity of directly searching for the most similar subtrajectory is $O(mn^3)$ (through traversal searching $\frac{n(n+1)}{2}$ subtrajectories of the data trajectory, and the time complexity of directly computing the similarity of two trajectories of length x and y by dynamic programming is $O(xy)$). Although a recent work [27] optimizes the time complexity of a single subtrajectory query problem from $O(mn^3)$ to $O(mn^2)$ through dynamic programming techniques, it is still unaffordable for most applications that need to find the optimal subtrajectory in a few seconds. In existing studies, only for *dynamic time wrapping distance* (DTW) and *Frechet distance* (FD), the similar subtrajectory search problem can be exactly solved in $O(nm)$ time complexity with particular algorithms [9, 20]. However, it cannot be extended to other trajectory distance functions.

In this paper, we propose the conversion-matching algorithm (CMA) to find the optimal subtrajectory by computing the minimum cost of converting the query trajectory into the data trajectory. With carefully tailored methods and transformation of the trajectory distance functions, we can incrementally fast track the optimal start position of the optimal subtrajectory in the data trajectory in $O(1)$ time. Given a query trajectory and a data trajectory, we search for the optimal subtrajectory with the time complexity of $O(nm)$. Meanwhile, the algorithm is applicable for the vast majority of distance functions. We use *weighted edit distance* (WED) [13] and *dynamic time warping* (DTW) [30] as examples to analyze the design of the algorithm. We also discuss how to apply our methods to other most popular trajectory distance functions. Experiments show that the performance of our algorithm is better than other existing methods.

To summarize, we make the following contributions:

- We propose CMA with the time complexity of $O(nm)$ to find the most similar subtrajectory for a query trajectory under most order-insensitive trajectory distance functions in Section 5.
- We describe the design idea of the algorithm in detail and simplify the calculation of conversion cost, using WED and DTW as examples in Section 6.
- We conduct experiments on three different real data sets to verify the superiority of our framework with the state-of-the-art similar subtrajectory query methods in Section 7.

The related work is discussed in Section 2. We formally define the similar subtrajectory search problem in Section 3. We also simply review the existing exact algorithms in Section 4.

2 RELATED WORK

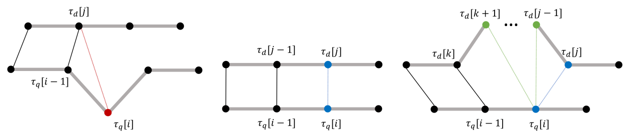
Trajectory Distance Function. Many works have proposed metrics to measure the distance between two trajectories [3, 5, 6, 13, 22, 24, 29–31]. We can divide these distance functions into two categories: order-insensitive and order-sensitive. The order-insensitive distance functions are independent of the position of the point in the trajectory; the order-sensitive distance functions are just the opposite. For example, the order-insensitive functions, DTW [30] and Frechet distance (FD) [3], define the distance between trajectories as the cost of turning one trajectory into another through substitution operations. DTW allows different points in one trajectory to be mapped to the same point in another trajectory, enabling DTW to deal well with the case where two trajectories are sampled at different frequencies. Compared with DTW, edit distance with real penalty (ERP) [5] introduces the insert and delete operations. The cost of inserting a point and deleting a point equals replacing it with a pre-defined default point. However, when the position of the default point is not set reasonably, the cost of deleting and inserting a point can be much greater than replacing it. Therefore, edit distance on real sequences (EDR) [6] fixes this issue by introducing an upper bound. Specifically, when the distance between a point in the trajectory and its replacement is greater than this upper bound, the replacement cost equals the deletion cost. WED is a generic distance function that allows users to customize the cost of deletion, insertion, and replacement. The order-sensitive distance functions (e.g., longest common subsequence (LCSS) [22], longest overlapping road segments (LORS) [24], and longest common road segments (LCRS) [31]) calculate the distance of a point in a trajectory from another trajectory considering the point positions in the trajectories.

Table 1: Summary of subtrajectory similarity search algorithms.

Algorithms	Accurate	DTW	ERP	EDR	FD
CMA (Ours)	exact	$O(mn)$	$O(mn)$	$O(mn)$	$O(mn)$
ExactS [27]	exact	$O(mn^2)$	$O(mn^2)$	$O(mn^2)$	$O(mn^2)$
Spring [20]	exact	$O(mn)$	-	-	-
Greedy Backtracking [9]	exact	-	-	-	$O(mn)$
POS [27]	approx.	$O(mn)$	$O(mn)$	$O(mn)$	$O(mn)$
PSS [27]	approx.	$O(mn)$	$O(mn)$	$O(mn)$	$O(mn)$
RLS [27]	approx.	$O(mn)$	$O(mn)$	$O(mn)$	$O(mn)$
RLS-Skip [27]	approx.	$O(mn)$	$O(mn)$	$O(mn)$	$O(mn)$

Subtrajectory Search. The previous work [13] divides the subtrajectory search into two stages: filtering and verification. In the filtering phase, most of the trajectories whose distance from the query trajectory exceeds a given threshold are filtered out to reduce the number of validations [8, 27]; in the validation phase, the execution time of the validation phase is simplified with the help of indexes. Unfortunately, this work invokes the trajectory distance function calculation method for all candidate subtrajectories within a trajectory during the validation phase, which makes the validation phase take much time. Another work [27] focuses on how to find the subtrajectory with the minimum distance from the query trajectory in the data trajectory given a query trajectory of length m and a data trajectory of length n. ExactS [27] is proposed to find the optimal subtrajectory in time complexity of $O(mn^2)$. Meanwhile, this work also proposes approximate algorithms (e.g., POS and PSS [27]) with $O(mn)$ time complexity. In addition to these traditional methods, this work proposes two reinforcement learning-based approximate methods (RLS, RLS-Skip [27]) to find the optimal subtrajectory. Furthermore, RLS and RLS-Skip can adaptively select appropriate split points to improve the efficiency of the search. With DTW as the distance function, Spring [20] can find the optimal subtrajectory exactly in $O(mn)$ time complexity. Besides, Greedy Backtracking [9] can find the exact optimal similar subtrajectory with $O(mn)$ time complexity on FD. However, Spring and Greedy Backtracking do not apply to other distance functions. In contrast, our CMA can be applied to most order-insensitive distance functions. Table 1 summarizes the existing subtrajectory search methods.

Applications of Subtrajectory search. Some previous studies [25, 28] implement the travel time estimation of a segment of the trajectory by a similar subtrajectory search. One specific process is to search the most similar subtrajectory from the database and then use its time as an estimate of the current trajectory's communication



(a) $Cost_{del}$ (b) $Cost_{sub}$ (c) $Cost_{ins}$

Figure 2: Demonstration of the conversion cost

time. The advantage of subtrajectory search is that it can solve the sparsity of trajectories in the database and thus find more similar trajectories. Another common application is to analyze the movement and behavioral performance of players on the sports ground through subtrajectory search [26]. In addition, subtrajectory search can be used to count the frequency of a given road section in the database for better road planning [7, 12, 16].

3 PROBLEM DEFINITION AND PRELIMINARIES

3.1 Basic Concepts

There are two types of trajectories for the SSS problem: query and data trajectories. We expect to search for the most similar subtrajectory for a given query trajectory under a specific distance function among a large volume of data trajectories. We first provide the definitions of trajectories and subtrajectories as follows:

Definition 1. (Trajectory) A trajectory τ with the length of n consists of a series of points denoted as $\langle p_1, p_2, p_3, \ldots, p_n \rangle$.

We denote the query trajectory as τ_q with the length of m and the data trajectory as τ_d with the length of n. The points of trajectories can be specific physical locations or nodes on a road network. In particular, we denote a trajectory without any points as $\tau_\emptyset$.

Definition 2. (Subtrajectory) Given a trajectory τ with the length of n, its subtrajectory is a portion of consecutive points, $\tau[i : j] = \langle p_i, p_{i+1}, \ldots, p_j \rangle$ $(1 \le i \le j \le n)$.

In particular, we denote the i^{th} point in τ as $\tau[i : i]$, abbreviated as $\tau[i]$. If $i > j$, we have $\tau[i : j] = \tau_\emptyset$.

Usually, we have a set of data trajectories. In this paper, we focus on finding an optimal subtrajectory from a data trajectory among many data trajectories to match the query trajectory. We have also implemented two pruning methods, Grid-Based Prune (GBP) and Key Point Filter (KPF), to help filter the irrelevant trajectories quickly. Please refer to our technical report for more details [1].

3.2 Distance Function

The distance function between trajectories represents the cost of transforming the points of the query trajectory into the data trajectory plus the cost of inserting prefix subtrajectory and suffix subtrajectory. We will give a detailed description of the matching and conversion costs next.

Definition 3 (Matching Sequence)**.** For a query trajectory, we define its matching sequence as $\mathcal{A}_{\tau_q:\tau_d} = \{a_1, a_2, a_3, \ldots, a_m\}$. For any $1 \le i \le m$, a_i indicates the index of $\tau_q[i]$'s matched point in the data trajectory (i.e., $\tau_q[i] = \tau_d[a_i]$). For any $i \le j$, we have $a_i \le a_j$.

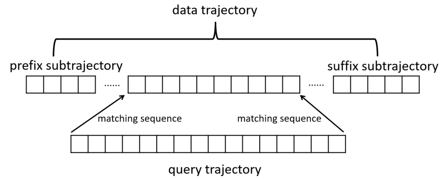


Figure 3: Demonstration of Matching Process

According to the definition, if a trajectory $\tau_d[s : t]$ is a subtrajectory of another $\tau_d[i : j]$, we have $\mathcal{A}_{\tau_q:\tau_d[s:t]} \subseteq \mathcal{A}_{\tau_q:\tau_d[i:j]}$. After defining the concept of matching, we can analyze the conversion cost *Cost* for each point in the query trajectory to its matching point in the data trajectory.

Conversion Cost. We assume that the current matching sequence is $\mathcal{A}_{\tau_q:\tau_d}$. When $\tau_q[i]$ matches $\tau_d[j]$, depending on the different matches of $\tau_q[i-1]$, we define three different conversion costs $Cost_{sub}$, $Cost_{ins}$ and $Cost_{del}$ and give the demonstration of these conversion costs in Figure 2:

(a) $a_{i-1} = j$. We need to remove $\tau_q[i]$, and denote the conversion cost as $Cost(\tau_q[i], \tau_d[a_i]) = Cost_{del}(\tau_q[i], \tau_d[j])$.

(b) $a_{i-1} = j-1$. In this case, we replace $\tau_q[i]$ with $\tau_d[j]$, and denote the conversion cost as $Cost(\tau_q[i], \tau_d[a_i]) = Cost_{sub}(\tau_q[i], \tau_d[j])$.

(c) $a_{i-1} = k$, where $1 \le k < j-1$. For this case, we substitute $\tau_q[i]$ with $\tau_d[j]$ and insert $\tau_d[k+1 : j-1]$. We denote the conversion cost as $Cost(\tau_q[i], \tau_d[a_i]) = Cost_{ins(k)}(\tau_q[i], \tau_d[j])$.

We can calculate the cost of transforming the query trajectory into a data trajectory for each matching sequence, which includes the cost of transforming each point in the query trajectory into a matching point in the data trajectory and the cost of inserting *prefix trajectory* and *suffix trajectory* of the data trajectory as shown in Figure 3. We denote the cost of inserting prefix trajectory and suffix trajectory as $Insert(\tau_d[1 : a_1 - 1]) + Insert(\tau_d[a_m + 1 : n])$. In particular, we consider the cost of inserting an empty trajectory to be 0, i.e., $Insert(\tau_\emptyset) = 0$. Given a matching sequence $\mathcal{A}_{\tau_q:\tau_d}$, its *matching cost* is $\sum_{a_i \in \mathcal{A}_{\tau_q:\tau_d}} Cost(\tau_q[i], \tau_d[a_i]) + Insert(\tau_d[1 : a_1 - 1]) + Insert(\tau_d[a_m + 1 : n])$.

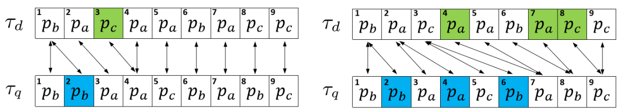
Definition 4 (Distance Function)**.** We denote the set of all feasible matching sequences between the query trajectory and the data trajectory as $\mathbb{A}$. Then, we define the distance $\Theta(\tau_q, \tau_d)$ between the query trajectory and the data trajectory as follows:

$$\begin{aligned} \Theta(\tau_q, \tau_d) &= \min_{\mathcal{A}_{\tau_q:\tau_d} \in \mathbb{A}} \sum_{a_i \in \mathcal{A}_{\tau_q:\tau_d}} Cost(\tau_q[i], \tau_d[a_i]) \\ &+ Insert(\tau_d[1 : a_1 - 1]) + Insert(\tau_d[a_m + 1 : n]) \end{aligned} \tag{1}$$

We define the *optimal matching sequence* as the matching sequence that minimizes the matching cost.

There are many trajectory distance functions, such as DTW [30], ERP [5], EDR [6], and WED [13]. In this paper, we use WED and DTW as examples to illustrate our definition.

WED. WED is a general distance function that allows the user-defined cost functions and contains several important cost functions (e.g., EDR and ERP). WED defines the distance $wed(\tau_q, \tau_d)$ between τ_q and τ_d as the minimum cost of converting τ_q to τ_d by a finite number of insertion, deletion and substitution. Given two points $\tau_q[i]$ and $\tau_d[j]$, we denote the cost of insertion, deletion and substitution by $ins(\tau_d[j])$, $del(\tau_q[i])$ and $sub(\tau_q[i], \tau_d[j])$. Besides, the cost of deleting the subtrajectory $\tau_q[i : j]$ and inserting the subtrajectory $\tau_d[i : j]$ are denoted as $del(\tau_q[i : j])$ and



(a) WED (b) DTW

Figure 4: Examples of Distance Function

$ins(\tau_d[i:j])$. We have $del(\tau_q[i:j]) = \sum_{i \le k \le j} = del(\tau_q[k])$ and $ins(\tau_d[i:j]) = \sum_{i \le k \le j} = ins(\tau_q[k])$.

Example 1. *Given two trajectories τ_q and τ_d as shown in Figure 4, we use WED to calculate the distance between them. The blue points indicate the deleted points in the transformation of τ_q into τ_d, while the green points indicate the inserted points. We set the cost of $ins(\tau_d[j])$, $del(\tau_q[i])$ to 1. In addition, we set the cost of $sub(\tau_q[i], \tau_d[j])$ to 1 if $\tau_q[i] \neq \tau_d[j]$; otherwise, it is set to 0. Figure 4(a) shows an optimal matching that converts τ_q into τ_d by deleting $\tau_q[2]$, inserting $\tau_d[3]$ and substituting $\tau_q[5]$ with $\tau_d[5]$ and $\tau_q[8]$ with $\tau_d[8]$. Since there are no redundant prefix and suffix subtrajectories, we have $Insert(\tau_d[1:a_1-1]) + Insert(\tau_d[a_m+1:n]) = 0$. Therefore, the distance between τ_q and τ_d is 4(= $del(\tau_q[2]) + ins(\tau_d[3]) + sub(\tau_q[5], \tau_d[5]) + sub(\tau_q[8], \tau_d[8])$).*

Moreover, we can compute the distance between τ_q and τ_d by a dynamic programming algorithm [12]. We have $wed(\tau_q[i:j], \tau_\emptyset) = del(\tau_q[i:j]) = \sum_{k=i}^{j} del(\tau_q[k])$ and $wed(\tau_\emptyset, \tau_d[i:j]) = ins(\tau_d[i:j]) = \sum_{k=i}^{j} ins(\tau_d[k])$. The $wed(\tau_q, \tau_d)$ is defined recursively:

$$
\begin{aligned}
& wed(\tau_q[1:i], \tau_d[1:j]) \\
= \; & \min \begin{cases} wed(\tau_q[1:i-1], \tau_d[1:j-1]) + sub(\tau_q[i], \tau_d[j]) \\ wed(\tau_q[1:i], \tau_d[1:j-1]) + ins(\tau_d[j]) \\ wed(\tau_q[1:i-1], \tau_d[1:j]) + del(\tau_q[i]) \end{cases}
\end{aligned}
$$

DTW. Another well-known distance function is DTW. Unlike WED, there is no deletion and insertion in DTW, instead, multiple points are allowed to be substituted for the same point in another trajectory. However, we try to interpret DTW from a different perspective to make it applicable to the algorithm proposed in this paper. We interpret the original substitution relation as a matching. We consider that only one point $\tau_q[i]$ is substituted for a point $\tau_d[j]$ in another trajectory, while other points that substitute $\tau_d[j]$ are deleted. We can define the insertion in the same way. The cost of deleting a point or inserting a point in the query trajectory is different, depending on which point it matches with, that is, $del(\tau_q[i]) = sub(\tau_q[i], \tau_d[j])$ and $ins(\tau_d[j]) = sub(\tau_q[i], \tau_d[j])$ if $\tau_q[i]$ matches $\tau_d[j]$.

Here, we give an example about the optimal matching when using DTW as distance function.

Example 2. *The optimal matching when converting the τ_q into τ_d is shown in Figure 4(b). We set the distance between two characters to 1 in case the two characters are not equal; otherwise, it is set to 0. The minimum cost to convert τ_q to τ_d is to delete $\tau_q[2], \tau_q[4], \tau_q[6]$ and insert $\tau_d[4], \tau_d[7], \tau_d[8]$. The cost of deleting $\tau_q[2]$ and $\tau_q[6]$ is different because $\tau_q[2]$ matches $\tau_d[1]$ while $\tau_q[6]$ matches $\tau_d[3]$ and $\tau_q[2] = \tau_d[1], \tau_q[6] \neq \tau_d[3]$. Thus, the distance between τ_q and τ_d is 2(= $del(\tau_q[2]) + del(\tau_q[4]) + del(\tau_q[6]) + ins(\tau_d[4]) + ins(\tau_d[7]) + ins(\tau_d[8]) = 0 + 0 + 1 + 0 + 1 + 0$) when using DTW as the distance function.*

Table 2: Symbols and Descriptions.

Symbol	Description
τ_d	the data trajectory
τ_q	the query trajectory
n	the length of data trajectory
m	the length of query trajectory
$\tau[i:j]$	a subtrajectory of τ from i^{th} point to j^{th} point
$\tau[i]$	the i^{th} point in trajectory τ
$\mathcal{A}_{\tau_q:\tau_d}$	a matching sequence between τ_q and τ_d
a_i	the matches of $\tau_q[i]$ and $\tau_d[a_i]$
Θ	the distance function

Finally, we also give the dynamic process for the calculation of $dtw(\tau_q, \tau_d)$ as follows:

$$
dtw(\tau_q[1:i], \tau_d[1:j]) = \begin{cases} \sum_{k=1}^{j} sub(\tau_q[1], \tau_d[k]), & i = 1 \\ \sum_{k=1}^{i} sub(\tau_q[k], \tau_d[1]), & j = 1 \\ \min\{dtw(\tau_q[1:i-1], \tau_d[1:j]), & \\ dtw(\tau_q[1:i], \tau_d[1:j-1]), & \\ dtw(\tau_q[1:i-1], \tau_d[1:j-1])\} & \\ +sub(\tau_q[i], \tau_d[j]), & else \end{cases} \tag{2}
$$

The algorithm proposed in this paper requires that the distance function to satisfy a specific property: the distance of points between different trajectories is independent of the position of the point in the trajectory. We will explain this in Section 6.3.

3.3 Problem Definition

We define the similar subtrajectory search problem as follows:

Definition 5 (Similar Subtrajectory Search Problem, SSS)**.** Given a query trajectory τ_q and a data trajectory τ_d, we expect a closest subtrajectory $\tau_d[i^* : j^*]$ under a given distance function Θ (e.g., WED or DTW) from the data trajectory for the query trajectory τ_q:

$$
(i^*, j^*) = \underset{1 \le i \le j \le n}{\arg\min}\, \Theta(\tau_q, \tau_d[i:j])
$$

A more general query is to find the *top-K* similar subtrajectories from massive data trajectories for the query trajectory. Instead, we can follow such a search process in previous work [26] that maintains the most similar K trajectories and updates it when a more similar subtrajectory appears. Therefore, we mainly consider querying the most similar subtrajectory the query trajectory.

Suppose the average length of a data trajectory is n, which means that a data trajectory has $\frac{n(n+1)}{2}$ subtrajectories. Assuming that the average length of a query trajectory is m and the complexity of computing the distance between the data trajectory and the query trajectory is $O(mn)$. Therefore, given query trajectory τ_q and data trajectory τ_d, the time complexity of searching a subtrajectory of τ_d with the smallest distance from τ_q in τ_d is $O(mn^3)$.

4 REVIEW OF EXISTING SOLUTIONS

We briefly review the existing exact algorithms for the SSS problem.

4.1 ExactS

The vast majority of distance functions [3, 5, 6, 13, 22, 24, 29–31] are defined via recursive processes. Using dynamic programming, we can compute the trajectory distance of a query trajectory and a subtrajectory of the data trajectory in $O(mn)$, where m and n are the

Algorithm 1: $ExactS(\tau_q, \tau_d)$ [27]

Input: a query trajectory τ_q, a data trajectory τ_d
Output: a subtrajectory $\tau_d[i^*, j^*]$

1 $i^* \leftarrow 0, j^* \leftarrow 0$
2 $score \leftarrow \infty$
3 **forall** $1 \le i \le n$ **do**
4 $\quad M \leftarrow DP(\tau_q, \tau_d[i:n])$
5 $\quad y^* \leftarrow \underset{1 \le y \le n-i+1}{\arg\min} M_{m,y}$
6 $\quad$ **if** $M_{i,y^*} < score$ **then**
7 $\quad\quad score \leftarrow M_{i,y^*}$
8 $\quad\quad i^* \leftarrow i$
9 $\quad\quad j^* \leftarrow y^* + i - 1$
10 **return** $\tau_d[i^*, j^*]$

lengths of the query trajectory and the data trajectory, respectively. For a query trajectory τ_q and a data trajectory τ_d, let $M_{x,y}$ denote the trajectory distance between $\tau_q[1:x]$ and $\tau_d[i:i+y]$ for a given iteration i. ExactS [27] can compute $M_{x,y}$ from $M_{x,y-1}$ using a dynamic programming technique. Thus, line 4 in Algorithm 1 can be solved in $O(mn)$. There are n iterations, thus the overall time complexity of ExactS is $O(mn^2)$. ExactS can be applied to most of the distance functions.

4.2 Spring

Spring algorithm [20] is based on the existing dynamic programming computational procedure of DTW and changes the initialization procedure of $dtw(\tau_q[1:i], \tau_d[1:j])$ in the Equation 2 when $i = 1$. Spring considers $\tau_d[1:j-1]$ to be redundant when $i = 1$; therefore, they modify the equation for $dtw(\tau_q[1:i], \tau_d[1:j])$ when $i = 1$ to be as follows:

$$dtw(\tau_q[1:i], \tau_d[1:j]) = sub(\tau_q[1], \tau_d[j]) \tag{3}$$

In addition, the authors demonstrate that a modification of the Equation 2 enables it to compute the optimal subtrajectory. However, this trick can only be applied to the DTW function and cannot be extended to other distance functions (e.g., ERP, EDR, and WED).

4.3 Greedy Backtracking

Greedy Backtracking [9] investigates finding the optimal subtrajectory in a data trajectory when using FD as the distance function. It constructs a matrix X, where $X_{i,j}$ denotes the Euclidean distance between $\tau_q[i]$ and $\tau_d[j]$. Assuming that $X_{1,1}$ denotes the upper left corner of the matrix, Greedy Backtracking finds a path from the top to the right or down until it reaches the bottom. The path's cost is the maximum value in the matrix through which the path passes, and Greedy Backtracking finds the optimal subtrajectory by finding the path with the lowest cost. Since FD only considers substitution operations between the trajectory point and trajectory point, it can construct the matrix S. However, the cost of converting $\tau_q[i]$ into $\tau_d[j]$ in other distance functions that consider insertion and deletion operations (e.g., ERP, EDR, and WED) is uncertain; thus, the matrix S cannot be constructed and Greedy Backtracking is not suitable.

5 CONVERSION-MATCHING ALGORITHM

This section presents an efficient and exact subtrajectory search algorithm, namely Conversion-Matching Algorithm (CMA). Firstly, we transform the problem of finding the optimal subtrajectory into a problem of finding the optimal matching sequence. Meanwhile, we introduce *the cost of optimal partial matching* $C_{i,j}$ to find the optimal matching sequence. Here, $C_{i,j}$ denotes the minimal cost of converting $\tau_q[1:i]$ into a subtrajectory of $\tau_d[1:j]$ when $\tau_q[i]$ matches $\tau_d[j]$ (i.e., $a_i = j$). Note that, converting $\tau_q[1:i]$ into $\tau_d[1:j]$ does not means that $\tau_q[1]$ must match $\tau_d[1]$. Finally, we propose the Conversion-Matching Algorithm (CMA) to calculate $C_{i,j}$ and find the optimal subtrajectory.

5.1 Optimal Matching Sequence

Although previous work [26] has optimized the time complexity of this problem to $O(mn^2)$, it still makes the computational cost increase dramatically when the length of the data trajectory is large. This paper reduces this time complexity to $O(mn)$ by a different dynamic programming algorithm based on a newly introduced concept.

Different from existing algorithms, the algorithm is not based on the existing dynamic programming method for calculating the distance. Instead, the basic idea of the algorithm is to calculate the minimum cost of converting the points in the query trajectory to the data trajectory by three operations: insertion, deletion and substitution. Each point in the query trajectory is converted to its matching point in the data trajectory at a specific cost in the conversion process. We can prove that the optimal subtrajectory do not contain redundant prefix trajectories and suffix trajectories by following theorem.

Theorem 5.1. *Assume that $\tau_d[i:j]$ is the optimal subtrajectory in τ_d, i.e., $\Theta(\tau_q, \tau_d[i:j]) = \min_{1 \le s \le t \le n} \Theta(\tau_q, \tau_d[s:t])$. Then, we have*

$$\Theta(\tau_q, \tau_d[i:j]) = \sum_{a_k \in \mathcal{A}^o_{\tau_q:\tau_d[i:j]}} Cost(\tau_q[k], \tau_d[a_k])$$

where $\mathcal{A}^o_{\tau_q:\tau_d[i:j]}$ is the optimal match sequence of τ_q and $\tau_d[i:j]$.

Proof. We will prove that $a_1 = i$ and $a_m = j$ in $\mathcal{A}_{\tau_q:\tau_d[i:j]}$ when $\tau_d[i:j]$ is the optimal subtrajectory.

Suppose $a_1 = s$ and $a_m = t$ ($s \ge i, t \le j$), then $\mathcal{A}^o_{\tau_q:\tau_d[i:j]}$ is also a matching sequence of $\tau_d[s:t]$. Therefore, we have

$$\begin{aligned}
\Theta(\tau_q, \tau_d[s:t]) \quad &\le \sum_{a_k \in \mathcal{A}^o_{\tau_q:\tau_d[i:j]} \setminus \{a_1, a_m\}} Cost(\tau_q[k], \tau_d[a_k]) \\
&\quad + Cost(\tau_q[1], \tau_d[s]) + Cost(\tau_q[m], \tau_d[t]) \\
&= \sum_{a_k \in \mathcal{A}^o_{\tau_q:\tau_d[i:j]}} Cost(\tau_q[k], \tau_d[a_k]) \\
&\le \Theta(\tau_q, \tau_d[i:j])
\end{aligned}$$

If $s > i$ or $t < j$, then $\tau_d[i:j]$ is not the optimal subtrajectory, which contradicts what is known. Therefore, we have $a_1 = i$ and

$a_m = j$. Further, we can obtain

$$\begin{aligned}\Theta(\tau_q, \tau_d[i:j]) &= \min_{\mathcal{A}_{\tau_q:\tau_d[i:j]} \in \mathbb{A}} \sum_{a_k \in \mathcal{A}_{\tau_q:\tau_d[i:j]}} Cost(\tau_q[k], \tau_d[a_k]) \\ &\quad + Insert(\tau_d[i:a_1-1]) + Insert(\tau_d[a_m+1:j]) \\ &= \sum_{a_k \in \mathcal{A}^o_{\tau_q:\tau_d[i:j]}} Cost(\tau_q[k], \tau_d[a_k]) \\ &\quad + Insert(\tau_d[i:a_1-1]) + Insert(\tau_d[a_m+1:j]) \\ &= \sum_{a_k \in \mathcal{A}^o_{\tau_q:\tau_d[i:j]}} Cost(\tau_q[k], \tau_d[a_k])\end{aligned}$$

□

Theorem 5.1 proves that we do not need to consider redundant prefix subtrajectory and suffix subtrajectory in the optimal subtrajectory problem but only need to consider minimizing the conversion cost of all matching points. Then, we will prove that the optimal matching sequence of optimal subtrajectory is also optimal among all matching sequences between query trajectory and data trajectory.

Theorem 5.2. *Assume that $\mathcal{A}^o_{\tau_q:\tau_d[i:j]}$ is the optimal matching sequence for the optimal subtrajectory $\tau_d[i:j]$, then it is also the optimal among all matching sequences, i.e.* $\mathcal{A}^o_{\tau_q:\tau_d[i:j]} = \arg\min_{\mathcal{A}_{\tau_q:\tau_d} \in \mathbb{A}} \sum_{a_i \in \mathcal{A}_{\tau_q:\tau_d}} Cost(\tau_q[i], \tau_d[a_i])$.

PROOF. We assume that the matching sequence $\mathcal{A}^p_{\tau_q:\tau_d}$ is better than $\mathcal{A}^o_{\tau_q:\tau_d[i:j]}$. If $\mathcal{A}^p_{\tau_q:\tau_d}$ is the matching sequence of subtrajectories $\tau_d[i:j]$ and query trajectories, then it contradicts the condition that $\mathcal{A}^o_{\tau_q:\tau_d[i:j]}$ is the optimal matching sequence for $\tau_d[i:j]$; conversely, if $\mathcal{A}^p_{\tau_q:\tau_d}$ is a matching sequence of the subtrajectory $\tau_d[a_1:a_m]$, then $\tau_d[i:j]$ is not an optimal subtrajectory, which contradicts what is known. □

By using the theorems 5.1 and 5.2, we can conclude

$$\begin{aligned}&\min_{1\le i\le j\le n} \Theta(\tau_q, \tau_d[i:j]) \\ = &\min_{\mathcal{A}_{\tau_q:\tau_d} \in \mathbb{A}} \sum_{a_i \in \mathcal{A}_{\tau_q:\tau_d}} Cost(\tau_q[i], \tau_d[a_i])\end{aligned} \tag{4}$$

According to the Equation 4, we reduce the problem of finding the optimal subtrajectory to finding the optimal matching sequence. We split all match sequences $\mathbb{A}$ of the query trajectory τ_q with the data trajectory τ_d according to the matches at different points. We use $\mathbb{A}[a_i = j]$ to denote the set of all matching sequences in $\mathbb{A}$ that satisfy the condition that $\tau_q[i]$ matches $\tau_d[j]$.

Definition 6 (Cost of Optimal Partial Matching)**.** We denote by $C_{i,j}$ the minimum value of the cost of converting $\tau_q[1:i]$ into a subtrajectory of $\tau_d[1:j]$ when $\tau_q[i]$ matches $\tau_d[j]$, that is,

$$C_{i,j} = \min_{\mathcal{A}_{\tau_q:\tau_d} \in \mathbb{A}[a_i=j]} \sum_{k=1, a_k \in \mathcal{A}_{\tau_q:\tau_d}}^{k=i} Cost(\tau_q[k], \tau_d[a_k])$$

Once we have calculated $C_{i,j}$, the distance between the query trajectory and the optimal subtrajectory is the minimum conversion cost when $\tau_q[m]$ matches a point in the data trajectory because

$$\begin{aligned}\min_{1\le i\le j\le n} \Theta(\tau_q, \tau_d[i:j]) &= \min_{\mathcal{A}_{\tau_q:\tau_d} \in \mathbb{A}} \sum_{a_i \in \mathcal{A}_{\tau_q:\tau_d}} Cost(\tau_q[i], \tau_d[a_i]) \\ = \min_{1\le j\le n} \min_{\mathcal{A}_{\tau_q:\tau_d} \in \mathbb{A}[a_m=j]} &\sum_{k=1, a_k \in \mathcal{A}_{\tau_q:\tau_d}}^{k=m} Cost(\tau_q[k], \tau_d[a_k]) \\ = \min_{1\le j\le n} C_{m,j}\end{aligned}$$

Therefore, we will mainly discuss how to compute $C_{i,j}$ in the subsequent section; meanwhile, we will use DTW and WED as examples to illustrate our algorithm in detail in Section 6.

5.2 Universal Calculation of $C_{i,j}$

This section will discuss how to calculate $C_{i,j}$ and find the subtrajectory with the shortest distance to the query trajectory from the data trajectory for a given query trajectory τ_q and a data trajectory τ_d.

Calculate $C_{i,j}$. We will discuss the computation process of $C_{i,j}$ in three cases:

1) $i = 1$. When $i = 1$, we substitute $\tau_q[1]$ with $\tau_d[j]$, which is $C_{i,j} = Cost_{sub}(\tau_q[i], \tau_d[j])$.

2) $j = 1$. There are two possible ways of converting $\tau_q[i]$ when $j = 1$: deleting $\tau_q[i]$, which means that $\tau_q[i-1]$ matches $\tau_d[1]$ so that we have $C_{i,j} = C_{i-1,j} + Cost_{del}(\tau_q[i], \tau_d[j])$; the other way is to substitute $\tau_q[i]$ with $\tau_d[1]$, which means that $\tau_q[1:i-1]$ will be deleted, resulting in $C_{i,j} = Cost_{sub}(\tau_q[i], \tau_d[j]) + \sum_{k=1}^{i-1} Cost_{del}(\tau_q[k], \tau_d[j])$. Therefore, we have $C_{i,j} = \min\{C_{i-1,j} + Cost_{del}(\tau_q[i], \tau_d[j]), Cost_{sub}(\tau_q[i], \tau_d[j]) + \sum_{k=1}^{i-1} Cost_{del}(\tau_q[k], \tau_d[j])\}$.

3) $1 < i \le m, 1 < j \le n$. Considering that the point $\tau_q[i]$ matches $\tau_d[j]$, there are three different conversion possibilities for $\tau_q[i]$ and $C_{i,j} = \min\{delCost_{i,j}, subCost_{i,j}, insCost_{i,j}\}$:

(a) $delCost_{i,j}$: deleting $\tau_q[i]$. When $\tau_q[i]$ is deleted, by the definition of matching, $\tau_q[i-1]$ and $\tau_d[j]$ are matched; thus, we have $delCost_{i,j} = C_{i-1,j} + Cost_{del}(\tau_q[i], \tau_d[j])$.

(b) $subCost_{i,j}$: substituting $\tau_q[i]$ with $\tau_d[j]$. In this case, $\tau_q[i-1]$ matches $\tau_d[j-1]$; thus, we have $subCost_{i,j} = C_{i-1,j-1} + Cost_{sub}(\tau_q[i], \tau_d[j])$.

(c) $insCost_{i,j}$: substituting $\tau_q[i]$ and inserting $\tau_d[k+1:j-1]$. In this situation, $\tau_q[i-1]$ may match $\tau_d[k]$ ($1 \le k < j-1$). We insert $\tau_d[k+1:j-1]$ and have $C_{i,j} = C_{i-1,k} + Cost_{ins(k)}(\tau_q[i], \tau_d[j])$. Considering all possible values of k, $insCost_{i,j} = \min_{1\le k<j-1} C_{i-1,k} + Cost_{ins(k)}(\tau_q[i], \tau_d[j])$.

In case 3.(b), our substitution of $\tau_q[i]$ for $\tau_d[j]$ can be seen as inserting an empty trajectory along with the substitution. Therefore, we will discuss 3.(b) and 3.(c) together in the subsequent sections.

To record the start position the optimal subtrajectory, we use $s_{i,j}$ to denote the index of $\tau_q[1]$'s matched point in τ_d, when $\tau_q[i]$ matches $\tau_d[j]$, i.e., the start position of the subtrajectory. Based on the computation process of $C_{i,j}$, we are able to determine which point $\tau_q[i-1]$ matches when $\tau_q[i]$ matches $\tau_d[j]$. Suppose $\tau_q[i-1]$ matches $\tau_d[k]$ ($1 \le k \le j$), then we have $s_{i,j} = s_{i-1,k}$. Finally, we propose CMA to solve the SSS problem as shown in Algorithm 2.

Complexity. Since $Cost_{sub}(\tau_q[i], \tau_d[j])$ and $Cost_{del}(\tau_q[i], \tau_d[j])$ involve only the substitution and deletion of one trajectory point, their time complexity is $O(1)$; therefore, when $i = 1$, the time complexity of $C_{i,j}$ is $O(1)$. We can calculate $\sum_{k=1}^{i-1} Cost_{del}(\tau_q[k], \tau_d[j])$ when $j = 1$ in advance for any i by preprocessing, and thus we can

Algorithm 2: $CMA(\tau_q, \tau_d)$

Input: a query trajectory τ_q, a data trajectory τ_d
Output: a subtrajectory $\tau_d[i^*, j^*]$

forall $1 \le i \le m$ **do**
  forall $1 \le j \le n$ **do**
    if $i = 1$ **then**
      $C_{i,j} \leftarrow Cost_{sub}(\tau_q[i], \tau_d[j])$
      $s_{i,j} \leftarrow j$
    else if $j = 1$ **then**
      $C_{i,j} \leftarrow \min\{C_{i-1,j} + Cost_{del}(\tau_q[i], \tau_d[j]), Cost_{sub}(\tau_q[i], \tau_d[j]) + \sum_{k=1}^{i-1} Cost_{del}(\tau_q[k], \tau_d[j])\}$
      $s_{i,j} \leftarrow 1$
    else
      $C_{i,j} \leftarrow \min\{delCost_{i,j}, subCost_{i,j}, insCost_{i,j}\}$
      update $s_{i,j}$ according to the matches of $\tau_q[i-1]$

$j^* \leftarrow \arg\min_{1 \le j \le n} C_m$
$i^* \leftarrow s_{m,j^*}$
return $\tau_d[i^*, j^*]$

compute $C_{i,j}$ within the time complexity of $O(1)$. In other cases, we need to calculate $\min\{delCost_{i,j}, subCost_{i,j}, insCost_{i,j}\}$. Given the specific distance function, we will discuss 6 how to compute $C_{i,j}$ in $O(1)$ time complexity in Section. Therefore, the time complexity of CMA is $O(mn)$.

6 FAST CALCULATION OF CONVERSION COSTS

In this section, we discuss how to calculate the conversion cost and $C_{i,j}$ for each point of the query trajectory with WED and DTW. Meanwhile, we will explain how $insCost_{i,j}$ can be computed in $O(1)$ time for the two distance functions WED and DTW.

6.1 Minimum Cost $C_{i,j}$ of WED

By introducing the concept of matching, we can convert the distance between trajectories into the cost required to convert points in τ_q into points in τ_d. Let's discuss the cost of converting each point $\tau_q[i]$ to its matched point $\tau_d[j]$ in τ_d.

Conversion Cost. There are three cases:

(a) $\tau_q[i-1]$ matches $\tau_d[j]$. We delete $\tau_q[i-1]$ so that $Cost_{del}(\tau_q[i], \tau_d[j]) = del(\tau_q[i])$.

(b) $\tau_q[i-1]$ matches $\tau_d[j-1]$. We substitute $\tau_q[i]$ with $\tau_d[j]$, i.e., $Cost_{sub}(\tau_q[i], \tau_d[j]) = sub(\tau_q[i], \tau_d[j])$.

(c) $\tau_q[i-1]$ matches $\tau_d[k]$, where $1 \le k < j-1$. The cost of converting $\tau_q[i]$ to $\tau_d[j]$ is the summation of $sub(\tau_q[i], \tau_d[j])$ and the cost of inserting the trajectory $\tau_d[k+1 : j-1]$. Therefore, we have $Cost_{ins(k)}(\tau_q[i], \tau_d[j]) = ins(\tau_d[k+1 : j-1]) + sub(\tau_q[i], \tau_d[j])$.

Example 3. *Consider the example in Figure 4, where τ_q is converted into τ_d. Since $\tau_q[1]$ has no predecessor node, $\tau_q[1]$ is only substituted for $\tau_d[1]$ with the cost of $sub(\tau_q[1], \tau_d[1])$. $\tau_q[2]$ matches $\tau_d[1]$, but since $\tau_q[1]$ matches $\tau_d[1]$, $\tau_q[2]$ has to be deleted, with the cost of $del(\tau_q[2])$. $\tau_q[4]$ matches $\tau_d[4]$ and $\tau_q[3]$ matches $\tau_d[2]$, thus $\tau_q[4]$ is converted to $\tau_d[4]$ with the cost of $sub(\tau_q[4], \tau_d[4]) + ins(\tau_d[3])$.*

Calculate $C_{i,j}$. After obtaining the conversion cost of the points using WED as a distance function, we can calculate $C_{i,j}$. We will discuss the relational equation for $C_{i,j}$ in three cases:
1) $i = 1$. $C_{i,j} = sub(\tau_q[1], \tau_d[j])$.
2) $j = 1$. $C_{i,j} = \min\{C_{i-1,j} + del(\tau_q[i]), sub(\tau_q[i], \tau_d[1]) + del(\tau_q[1 : i-1])\}$.
3) $1 < i \le m, 1 < j \le n$. Considering that the point $\tau_q[i]$ matches $\tau_d[j]$, $\tau_q[i]$ may need to be deleted or substituted. Thus, the update of $C_{i,j}$ depends mainly on whether $\tau_q[i]$ is deleted or replaced:

(a) $delCost_{i,j}$. When $\tau_q[i]$ is deleted, by the definition of matching, $\tau_q[i-1]$ and $\tau_d[j]$ are matched and we have $delCost_{i,j} = C_{i-1,j} + del(\tau_q[i])$.

(b) $subCost_{i,j}$ and $insCost_{i,j}$. $\tau_q[i-1]$ may match $\tau_d[k]$ ($1 \le k < j-1$) while substituting $\tau_q[i]$. In this case, we insert $\tau_d[k+1 : j-1]$ and have $C_{i,j} = C_{i-1,k} + ins(\tau_d[k+1 : j-1]) + sub(\tau_q[i], \tau_d[j])$. Considering all possible values of k, $insCost_{i,j} = \min_{1 \le k < j-1} C_{i-1,k} + ins(\tau_d[k+1 : j-1]) + sub(\tau_q[i], \tau_d[j])$. Another situation is that $\tau_q[i-1]$ matches $\tau_d[j-1]$ and we have $subCost_{i,j} = C_{i-1,j-1} + sub\left(\tau_q[i], \tau_d[j]\right)$. Combining these two situation, we have $C_{i,j} = \min\{insCost_{i,j}, subCost_{i,j}\} = \min_{1 \le k < j} C_{i-1,k} + ins(\tau_d[k+1 : j-1]) + sub(\tau_q[i], \tau_d[j])$. Then, the calculation of $C_{i,j}$ can be simplified by follows:

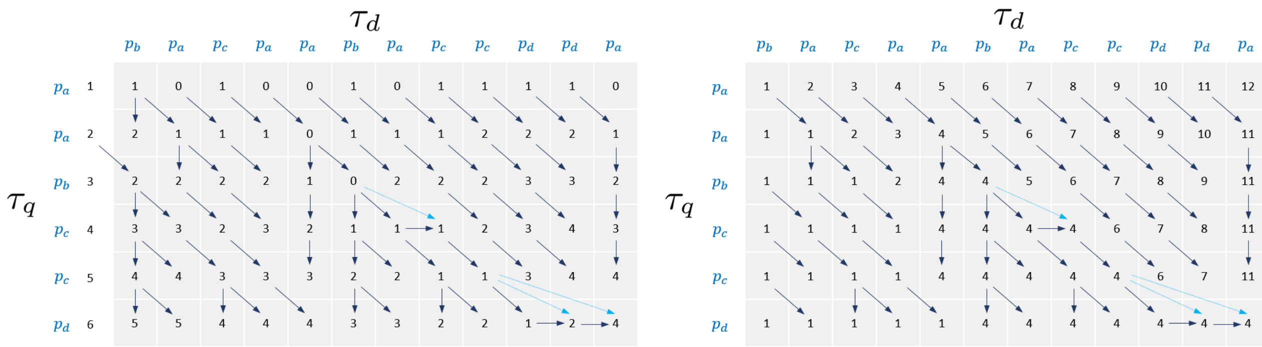
$$
\begin{aligned}
C_{i,j} = &\min_{1 \le k < j} C_{i-1,k} + ins\left(\tau_d[k+1 : j-1]\right) \\
&+ sub\left(\tau_q[i], \tau_d[j]\right) \\
&= \min\{\min_{1 \le k < j-1} C_{i-1,k} + ins\left(\tau_d[k+1 : j-1]\right) \\
&+ sub\left(\tau_q[i], \tau_d[j]\right), C_{i-1,j-1} + sub\left(\tau_q[i], \tau_d[j]\right)\} \\
&= \min\{C_{i,j-1} + ins\left(\tau_d[j-1]\right) - sub\left(\tau_q[i], \tau_d[j-1]\right) \\
&+ sub\left(\tau_q[i], \tau_d[j]\right), C_{i-1,j-1} + sub\left(\tau_q[i], \tau_d[j]\right)\}
\end{aligned}
$$

By the above analysis, we can obtain the expression for the calculation of $C_{i,j}$ while using WED as distance function

$$
C_{i,j} = \begin{cases}
sub(\tau_q[i], \tau_d[j]), & i = 1 \\
\min\{C_{i-1,j} + del(\tau_q[i]), sub(\tau_q[i], \tau_d[1]) & \\
+del(\tau_q[1 : i-1])\}, & j = 1, i \ne 1 \\
\min\{C_{i-1,j} + del(\tau_q[i]), & \\
C_{i,j-1} + ins\left(\tau_d[j-1]\right) - & \\
sub\left(\tau_q[i], \tau_d[j-1]\right) + sub\left(\tau_q[i], \tau_d[j]\right), & \\
C_{i-1,j-1} + sub\left(\tau_q[i], \tau_d[j]\right)\}, & otherwise
\end{cases} \tag{5}
$$

Finally, we illustrate algorithm with an example as follows:

Example 4. *Given two trajectories as shown in Figure 5, we need to find the subtrajectory from τ_d that is closest to τ_q. The insertion, deletion, and substitution costs are the same as the settings in Example 2. At the beginning, we will initialize $C_{1,j}$ (i.e., $C_{1,j} = sub(\tau_q[1], \tau_d[j])$). Then initialize $C_{i,1}$ based on whether $\tau_q[i]$ matches $\tau_d[1]$. Figure 5(a) shows that $\tau_d[1] = b$, thus only $\tau_q[3]$ is substituted with it. For $\tau_q[4]$, when it matches $\tau_d[8]$, we need to determine which point is optimal for $\tau_q[3]$ to match with. From Figure 5(a), we can see that the cost of $\tau_q[3]$ when matching with $\tau_d[6]$ is 0, being the minimum, which means we need to insert $\tau_d[7]$. Therefore, considering $\tau_q[4] = \tau_d[8]$, we can compute the result of $C_{4,8}$ from $C_{3,6}$, i.e., $C_{4,8} = C_{3,6} + ins(\tau_d[7]) + sub(\tau_q[4], \tau_d[8]) = 0 + 1 + 0 = 1$. In the actual implementation of the algorithm 2, we will compute $C_{4,8}$ by $C_{4,7}$, i.e. $C_{4,8} = C_{4,7} + ins(\tau_d[7]) - sub(\tau_q[4], \tau_d[7]) + sub(\tau_q[4], \tau_d[8]) = 1 + 1 - 1 + 0 = 1$.*



(a) $C_{i,j}$

(b) $S_{i,j}$

Figure 5: Demonstration of Calculating $C_{i,j}$ and $S_{i,j}$ when using WED as distance function

On the other hand, the algorithm updates $S_{i,j}$ as it executes. For example, when $\tau_q[4]$ matches $\tau_d[8]$, $\tau_q[3]$ is matched with $\tau_d[6]$ and we have $S_{4,8} = S_{3,6}$ as shown in Figure 5(b).

6.2 Minimum Cost $C_{i,j}$ of DTW

Unlike WED, the cost required to delete a point or insert a point in DTW is different. Firstly, we analyze the cost of converting each point $\tau_q[i]$ in the query trajectory to its matched point $\tau_d[j]$ in the data trajectory.

Conversion Cost. There are three cases:

(a) $\tau_q[i-1]$ matches $\tau_d[j]$. The cost of deleting $\tau_q[i]$ is equal to the cost of substituting $\tau_q[i]$ with $\tau_d[j]$, thus $Cost_{del}(\tau_q[i], \tau_d[j]) = sub(\tau_q[i], \tau_d[j])$.

(b) $\tau_q[i-1]$ matches $\tau_d[j-1]$. We substitute $\tau_q[i]$ with $\tau_d[j]$, i.e., $Cost_{sub}(\tau_q[i], \tau_d[j]) = sub(\tau_q[i], \tau_d[j])$.

(c) $\tau_q[i-1]$ matches $\tau_d[k]$, where $1 \le k < j-1$. The cost of converting $\tau_q[i]$ to $\tau_d[j]$ is the summation of $sub(\tau_q[i], \tau_d[j])$ and the cost of inserting the trajectory $\tau_d[k+1 : j-1]$. The cost for inserting subtrajectories $\tau_d[k+1 : j-1]$ depends on the points matched by $\tau_d[k+1 : j-1]$ at τ_q. Suppose $\tau_q[i-1]$ matches $\tau_d[k]$ and $\tau_q[i]$ will be matched into $\tau_d[j]$. Thus, the cost to insert $\tau_d[k+1 : j-1]$ is $\min\limits_{k \le t \le j-1} \sum_{p=k+1}^{t} sub(\tau_q[i-1], \tau_d[p]) + \sum_{p=t+1}^{j-1} sub(\tau_q[i], \tau_d[p])$. Thus, we have $Cost_{ins(k)}(\tau_q[i], \tau_d[j]) = \min\limits_{k \le t \le j-1} \sum_{p=k+1}^{t} sub(\tau_q[i-1], \tau_d[p]) + \sum_{p=t+1}^{j-1} sub(\tau_q[i], \tau_d[p]) + sub(\tau_q[i], \tau_d[j])$.

Example 5. *Let's take Figure 4(b) as an example, the cost of converting $\tau_q[1]$ to $\tau_d[1]$ when $i = 1$ and $j = 1$ is $sub(\tau_q[1], \tau_d[1]) = sub(b, b)$. When $i = 2$ and $j = 1$, $\tau_q[2]$ can only be converted to $\tau_d[1]$, thus the cost of the conversion is $sub(\tau_q[2], \tau_d[1]) = sub(b, b)$. By the time $\tau_q[4]$ matches $\tau_q[2]$, we need to delete $\tau_q[4]$ requiring a cost of $del(\tau_q[4]) = sub(\tau_q[4], \tau_q[2])$ because $\tau_q[3]$ matches $\tau_q[2]$. For i=9 and j=9, since $\tau_q[8]$ matches $\tau_d[7]$, the cost of converting $\tau_q[9]$ to $\tau_d[9]$ consists of not only the cost of the substitution $sub(\tau_q[9], \tau_d[9])$, but also the cost of inserting $\tau_d[8]$, that is, $\min\limits_{7 \le t \le 8} \sum_{p=8}^{t} sub(\tau_q[8], \tau_d[p]) + \sum_{p=t+1}^{8} sub(\tau_q[9], \tau_d[p])$. It is equal to $\min\{sub(\tau_q[8], \tau_d[8]), sub(\tau_q[9], \tau_d[8])\}$. It can be understood in another way that when $\tau_q[8]$ matches $\tau_d[7]$ and $\tau_q[9]$ matches $\tau_d[9]$, inserting $\tau_d[8]$ is equivalent to replacing $\tau_d[8]$ with $\tau_q[8]$ or $\tau_q[9]$.*

Calculate $C_{i,j}$. After analyzing the conversion cost, similarly, we discuss the computation of $C_{i,j}$ in three cases:

(a) $i = 1$. When $i = 1$, $\tau_q[1]$ can only be substituted with $\tau_d[j]$ as the same as WED and we have $C_{i,j} = sub(\tau_q[1], \tau[j])$.

(b) $j = 1$. Considering that the cost of deleting $\tau_q[i]$ and substituting $\tau_q[i]$ is the same when $j = 1$, we have

$$
\begin{aligned}
C_{i,j} &= \min\{Cost_{sub}(\tau_q[i], \tau_d[j]) + \sum_{k=1}^{i-1} Cost_{del}(\tau_q[k], \tau_d[j]), \\
&\quad C_{i-1,j} + Cost_{del}(\tau_q[i], \tau_d[j])\} \\
&= \min\{\sum_{k=1}^{i} sub(\tau_q[k], \tau_d[j]), C_{i-1,j} + sub(\tau_q[i], \tau_d[1])\} \\
&= C_{i-1,j} + sub(\tau_q[i], \tau_d[1])
\end{aligned}
$$

(c) $1 < i \le m, 1 < j \le n$. If we delete $\tau_q[i]$, we have $delCost_{i,j} = C_{i-1,j} + sub(\tau_q[i], \tau_q[j])$. Another conversion is substitution. $\tau_q[i-1]$ may be matched with any $\tau_d[k]$ $(1 \le k < j)$, and $C_{i,j}$ denotes the smallest of all possible values. Thus, we have

$$
\begin{aligned}
C_{i,j} &= \min\{insCost_{i,j}, subCost_{i,j}\} \\
&= \min\{\min_{1 \le k < j-1} C_{i-1,k} + Cost_{ins(k)}(\tau_q[i], \tau_d[j]), \\
&\quad C_{i-1,j-1} + Cost_{sub}(\tau_q[i], \tau_d[j])\} \\
&= \min_{1 \le k < j} C_{i-1,k} + \min_{k \le t < j-1} \sum_{p=k+1}^{t} sub(\tau_q[i-1], \tau_d[p]) \\
&\quad + \sum_{p=t+1}^{j-1} sub(\tau_q[i], \tau_d[p]) + sub(\tau_q[i], \tau_d[j]) \\
&= \min_{1 \le k < j} \min_{k \le t < j-1} C_{i-1,k} + \sum_{p=k+1}^{t} sub(\tau_q[i-1], \tau_d[p]) \\
&\quad + \sum_{p=t+1}^{j-1} sub(\tau_q[i], \tau_d[p]) + sub(\tau_q[i], \tau_d[j])
\end{aligned}
$$

The time complexity of computing $C_{i,j}$ $(1 < i < m, 1 < j < n)$ directly from the above expression is very high, and therefore we are required to simplify the computation of $C_{i,j}$ by Theorem 6.1.

Theorem 6.1. *When $i \ge 2, j \ge 2$, we have $C_{i,j} = \min\limits_{1 \le k < j} C_{i-1,k} + \sum_{t=k+1}^{j} sub(\tau_q[i], \tau_d[t])$.*

PROOF. We use mathematical induction to prove this theorem. To simplify the proof, we denote $\sum_{t=k+1}^{j} sub(\tau_q[i], \tau_d[t])$ as $sub(i, k+$

$1:j$). For $\forall j \geq 2$ when $i=2$, we have

$$\begin{aligned} C_{2,j} &= \min_{1\leq k<j}\min_{k\leq t<j} C_{1,k} + sub(1,k+1:t) + sub(2,t+1:j) \\ &= \min_{1\leq t<j}\min_{1\leq k\leq t} sub(1,k:t) + sub(2,t+1:j) \\ &= \min_{1\leq t<j} sub(\tau_q[1],\tau_d[t]) + sub(2,t+1:j) \\ &= \min_{1\leq t<j} C_{1,t} + \sum_{k=t+1}^{j} sub(\tau_q[2],\tau_d[k]) \\ &= \min_{1\leq k<j} C_{1,k} + \sum_{t=k+1}^{j} sub(\tau_q[2],\tau_d[t]) \end{aligned}$$

Suppose $i=h-1$, and we have $C_{h-1,j} = \min_{1\leq k<j} C_{h-2,k} + \sum_{t=k+1}^{j} sub(\tau_q[h-1],\tau_d[t]) = \min_{1\leq k<j} C_{h-2,k} + sub(h-1,k+1:j)$. Next, we have to prove that the theorem also holds when $i=h$.

$$\begin{aligned} C_{h,j} &= \min_{1\leq k<j}\min_{k\leq t<j} C_{h-1,k} + sub(h-1,k+1:t) + sub(h,t+1:j) \\ &= \min_{1\leq t<j}\min_{1\leq k\leq t} C_{h-1,k} + sub(h-1,k+1:t) + sub(h,t+1:j) \\ &= \min_{1\leq t<j}\min_{1\leq k\leq t}\min_{1\leq l<k} C_{h-2,l} + sub(h-1,l+1:k) \\ &\quad + sub(h-1,k+1:t) + sub(h,t+1:j) \\ &= \min_{1\leq t<j}\min_{1\leq l<t} C_{h-2,l} + sub(h-1,l+1:t) + sub(h,t+1:j) \\ &= \min_{1\leq t<j} C_{h-1,t} + sub(h,t+1:j) \\ &= \min_{1\leq k<j} C_{h-1,k} + \sum_{t=k+1}^{j} sub(\tau_q[h],\tau_d[t]) \end{aligned}$$

The above analysis shows that the theorem holds when $i=2$ and the theorem holds when $i=h-1$ can infer that the theorem holds when $i=h$. Therefore, the theorem holds. □

After obtaining the expression for $C_{i,j}$ from the theorem 6.1, we can further simplify it.

$$\begin{aligned} C_{i,j} &= \min_{1\leq k<j} C_{i-1,k} + \sum_{t=k+1}^{j} sub(\tau_q[i],\tau_d[t]) \\ &= \min\{\min_{1\leq k<j-1} C_{i-1,k} + \sum_{t=k+1}^{j-1} sub(\tau_q[i],\tau_d[t]), \\ &\quad C_{i-1,j-1}\} + sub(\tau_q[i],\tau_d[j]) \\ &= \min\{C_{i,j-1}, C_{i-1,j-1}\} + sub(\tau_q[i],\tau_d[j]) \end{aligned}$$

Finally, integrating all the previous analysis results, we can get the computational expression of $C_{i,j}$. With the Equation 6, we can quickly adapt the Algorithm 2 to get the optimal subtrajectory using DTW as the distance function.

$$C_{i,j} = \begin{cases} sub(\tau_q[i],\tau_d[j]), & i=1 \\ C_{i-1,j} + sub(\tau_q[i],\tau_d[1]), & j=1, i\neq 1 \\ \min\{C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}\} & \\ +sub\left(\tau_q[i],\tau_d[j]\right), & otherwise \end{cases} \tag{6}$$

6.3 Other Similarity Functions

In addition to DTW and WED, our method is also valid for other order-insensitive distance functions. EDR and ERP are specific cases of WED functions. Therefore, we only need to define sub, ins, and del in Equation 5. We denote the euclidean distance between two points $\tau_q[i]$ and $\tau_d[j]$ as $d(\tau_q[i],\tau_d[j])$. We can convert WED to ERP and EDR by defining sub, ins and del: (i) ERP. We can convert WED into ERP by making $sub(\tau_q[i],\tau_d[j]) = d(\tau_q[i],\tau_d[j])$, $del(\tau_q[i]) = d(\tau_q[i],q_c)$, $ins(\tau_d[j]) = d(\tau_d[j],q_c)$, where q_c is a fixed point on the map (e.g., the center of the region). (ii) EDR. $ins(\tau_d[j])$ and $del(\tau_q[i])$ in EDR are both 1, while $sub(\tau_q[i],\tau_d[j])$ takes a value of 0 if and only if $d(\tau_d[j],q_c) < \epsilon$ holds; otherwise, $sub(\tau_q[i],\tau_d[j]) = 1$.

FD is similar to DTW. In the same way, we can obtain the expressions for $C_{i,j}$ when FD is the distance function.

$$C_{i,j} = \begin{cases} sub\left(\tau_q[i],\tau_d[j]\right), & i=1 \\ \max\left\{C_{i-1,j}, sub(\tau_q[i],\tau_d[1])\right\}, & j=1, i\neq 1 \\ \max\{\min\{C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}\}, & \\ sub\left(\tau_q[i],\tau_d[j]\right)\}, & otherwise \end{cases} \tag{7}$$

When the order-insensitive distance functions are used as the trajectory distance functions, the calculation of the conversation cost does not depend on the position of the current point in the trajectory.

Unfortunately, our method does not apply to the subtrajectory search problem when order-sensitive trajectory distance function (e.g., LCSS) is used as the trajectory distance function. The reason is that we do not consider the position from which the subtrajectory starts when computing $C_{i,j}$. When $\tau_q[i]$ matches $\tau_d[j]$, the cost of converting $\tau_q[i]$ to $\tau_d[j]$ is only related to the matching relationship between $\tau_q[i-1]$ and $\tau_d[k]$ $(1\leq k\leq j)$. However, when LCSS is used as the distance function, the cost of converting $\tau_q[i]$ to $\tau_d[j]$ is also related to the matching relation of $\tau_q[1]$, i.e., the starting position of the subtrajectory. The starting position of the subtrajectories has a great influence on judging the distance between the points in two trajectories when LCSS is used as the distance function. Therefore, our algorithm is not suitable for a class of distance functions that considers the position of points in the trajectory, such as LCSS.

Table 3: Dataset statistics.

dataset	Xi'an	Chengdu	Porto
trajectories	111102	198031	1294719
average length	261	175	42
interval (s)	3	3	15
δ	10	10	2
q	70,**90**,110,130,150	30,**50**,70,90,110	6,**10**,14,18
r	0.05	0.05	0.20

7 EXPERIMENTAL STUDY

7.1 Data Set

We conduct experiments on three real data sets: (i) Chengdu Taxi Trip Dataset. DiDi Chuxing GAIA Open Dataset [17] provides taxi trips in Chengdu, China. We use the taxi trip records on November 1st. The size of Chengdu is $8.69km \times 8.04km$ (i.e., 104.04°∼104.13°, 30.65°∼30.73°). (ii) Xi'an Taxi Trip Dataset. DiDi Chuxing GAIA Open Dataset [17] also provides a dataset of taxi trips in Xi'an, China. We use the taxi trip records on October 1st, containing 111,102 trajectories. The size of Xi'an is $11.0km \times 8.6km$ (i.e., 108.91°∼109.00°, 34.20°∼34.29°). (iii) Porto Taxi Trip Dataset. Porto is a dataset describing a whole year (i.e., from July 1st, 2013 to June 30th, 2014) of the trajectories for all the 442 taxis running in the city of Porto, in Portugal [18]. We selected an area of $8.34km \times 8.85km$ (−8.67°∼−8.57°, 41.11°∼41.19°) near the center of Porto. Meanwhile, we clean the trajectories.

Table 4: Effectiveness of Algorithms.

distance function		DTW			EDR			ERP			FD		
dataset	algorithm	AR	MR	RR	AR	MR	RR	AR	MR	RR	AR	MR	RR
Xian	POS	35.563473	10505.00	18.12%	1.516196	286.84	1.14%	1.529726	36.33	0.15%	21.581594	3970.00	5.59%
	PSS	4.374571	676.99	2.99%	1.460815	378.33	1.34%	1.793466	43.67	0.17%	1.456668	27.27	0.03%
	RLS	3.803218	538.45	2.38%	1.434072	304.03	1.08%	1.564813	34.32	0.14%	1.389806	22.68	0.02%
	RLS-Skip	7.703218	1649.57	4.48%	1.536443	370.60	1.33%	1.691469	41.94	0.17%	3.531339	434.34	0.60%
	MA	**1**	**1**	**0%**	**1**	**1**	**0%**	**1**	**1**	**0%**	**1**	**1**	**0%**
	ExactS	1	1	0%	1	1	0%	1	1	0%	1	1	0%
Chengdu	POS	23.042967	3216.08	16.90%	1.615970	285.20	1.28%	1.490315	21.49	0.23%	13.194228	1029.49	3.70%
	PSS	4.902728	345.92	3.05%	1.598859	376.55	3.83%	2.593796	58.05	0.55%	1.362995	109.32	0.37%
	RLS	4.237570	275.23	2.43%	1.549786	302.60	3.08%	2.177677	45.56	0.43%	1.313524	90.32	0.30%
	RLS-Skip	6.904767	629.04	4.41%	1.677076	368.83	3.59%	2.377169	53.13	0.51%	2.591752	207.45	0.72%
	MA	**1**	**1**	**0%**	**1**	**1**	**0%**	**1**	**1**	**0%**	**1**	**1**	**0%**
	ExactS	1	1	0%	1	1	0%	1	1	0%	1	1	0%
Porto	POS	3.033461	351.01	13.09%	1.432727	321.91	15.35%	1.577079	62.09	1.92%	3.090780	221.96	5.29%
	PSS	1.976035	128.91	6.80%	1.352727	237.01	9.11%	2.665266	146.42	6.01%	1.450533	162.63	2.74%
	RLS	1.830677	102.69	5.40%	1.343469	190.54	7.32%	2.232406	114.62	4.70%	1.384809	134.26	2.25%
	RLS-Skip	2.140050	150.19	7.38%	1.425769	246.45	9.77%	2.447045	134.79	5.48%	1.643495	173.68	3.08%
	MA	**1**	**1**	**0%**	**1**	**1**	**0%**	**1**	**1**	**0%**	**1**	**1**	**0%**
	ExactS	1	1	0%	1	1	0%	1	1	0%	1	1	0%

Table 5: Efficiency of Algorithms.

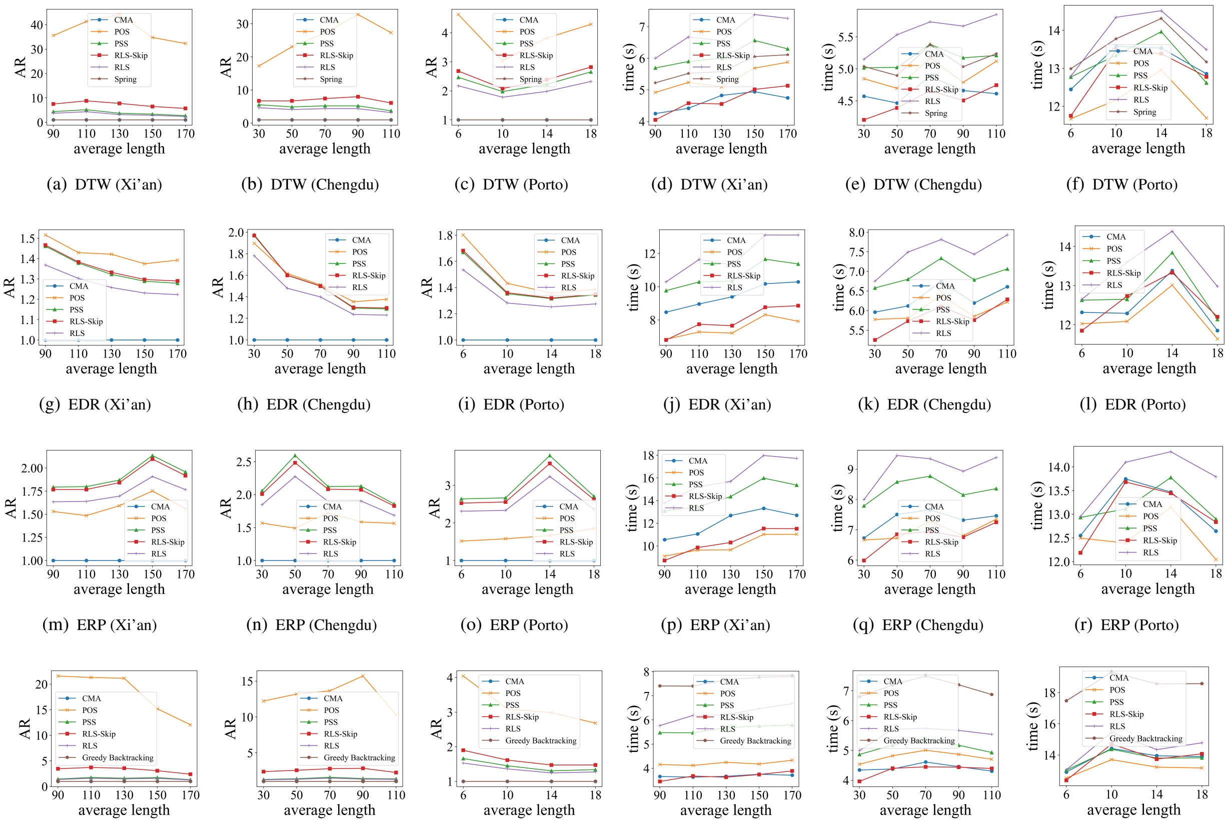
distance function		Time Cost (s)			
dataset	algorithm	DTW	EDR	ERP	FD
Xian	POS	4.99	**7.06**	9.42	4.03
	PSS	5.70	9.62	12.65	5.33
	RLS	6.00	10.82	13.92	5.87
	RLS-Skip	**4.19**	7.36	**9.32**	3.55
	CMA	4.25	8.04	10.15	**3.48**
	ExactS	1159.14	2032.44	2601.42	943.96
Chengdu	POS	4.31	**5.57**	6.57	4.62
	PSS	5.30	6.99	8.99	5.20
	RLS	5.59	7.86	9.89	5.72
	RLS-Skip	**3.99**	5.74	**6.55**	**4.25**
	CMA	4.47	6.22	7.19	4.34
	ExactS	769.36	1039.80	1211.19	780.45
Porto	POS	12.02	12.00	**12.41**	13.95
	PSS	12.69	12.23	12.94	14.85
	RLS	13.36	13.75	14.24	16.34
	RLS-Skip	**11.68**	**11.72**	12.64	**13.94**
	CMA	13.58	12.13	14.10	15.32
	ExactS	567.30	509.24	589.12	640.91

In this experiment, we generate Q query trajectories with lengths in the interval $[q-\delta, q+\delta]$ from all trajectories with the given values q and δ and take the average of the results. Specifically, we randomly select Q trajectories with lengths greater than $q-\delta$ and smaller than $q+\delta$ as query trajectory, while the other trajectories are used as data trajectories. For different datasets, we pick different q and δ. We set Q to 100 by default. Finally, we present the statistical information of the trajectory data in Table 3, where the default values of parameters are in bold font.

7.2 Approaches and Measurements

Search Algorithm. We mainly compare our algorithm CMA with the following existing methods:
1) ExactS. When it computes the distance between the query trajectory and some subtrajectories of the data trajectory, it records these intermediate results. Then, ExactS can utilize a dynamic programming technique to optimize the time complexity of searching the optimal subtrajectory from a data trajectory to $O(mn^2)$.
2) PSS and POS. The main idea of PSS is to traverse each point of a data trajectory to find the appropriate splitting position. The current optimal subtrajectory is updated by comparing the distance between the subtrajectory before the splitting point and the subtrajectory after the splitting point and the query trajectory. Then, the next suitable splitting point is found starting from the current splitting point. PSS can find an approximate solution of the optimal subtrajectory within the time complexity of $O(mn)$. As a variant of PSS, POS does not consider the subtrajectory after the splitting point. Therefore, the efficiency of POS is substantially improved compared with PSS, but the result quality of PSS is better than that of POS.
3) RLS and RLS-Skip. RLS is an algorithm based on reinforcement learning to determine whether to split the current point, and RLS takes a different action based on the state of the current point. RLS-Skip, on the other hand, adds a new action to RLS by not segmenting the current point and skipping the next point to traverse the entire trajectory faster. As a result, RLS-Skip can get a solution in less time, while RLS can find a better solution.
4) Spring and Greedy Backtracking. Both algorithms are of time complexity $O(mn)$. However, unlike the method proposed in this paper, Spring and Greedy Backtracking can only be applied to specific distance functions, DTW and FD, respectively. Therefore, we will test the performance and results of these two algorithms under particular functions in different data sets.

Considering that there are a large number of data trajectories in the database, to improve the efficiency of searching the optimal subtrajectories, we use the pruning methods in the subsequent experiments to filter out the data trajectories that are different from the query trajectories. This paper proposes two modules to filter data trajectories that are not similar to the query trajectory: Filter with Key Points (FKP) and Grid-Based Pruning (GBP). They are compared with the SOTA pruning method, OSF [13]. The details of



(a) DTW (Xi'an) (b) DTW (Chengdu) (c) DTW (Porto) (d) DTW (Xi'an) (e) DTW (Chengdu) (f) DTW (Porto)

(g) EDR (Xi'an) (h) EDR (Chengdu) (i) EDR (Porto) (j) EDR (Xi'an) (k) EDR (Chengdu) (l) EDR (Porto)

(m) ERP (Xi'an) (n) ERP (Chengdu) (o) ERP (Porto) (p) ERP (Xi'an) (q) ERP (Chengdu) (r) ERP (Porto)

(s) FD (Xi'an) (t) FD (Chengdu) (u) FD (Porto) (v) FD (Xi'an) (w) FD (Chengdu) (x) FD (Porto)

Figure 6: Effectiveness and efficiency with varying query lengths

the pruning methods and their experimental results can be found in the Appendix of our technical report [1].

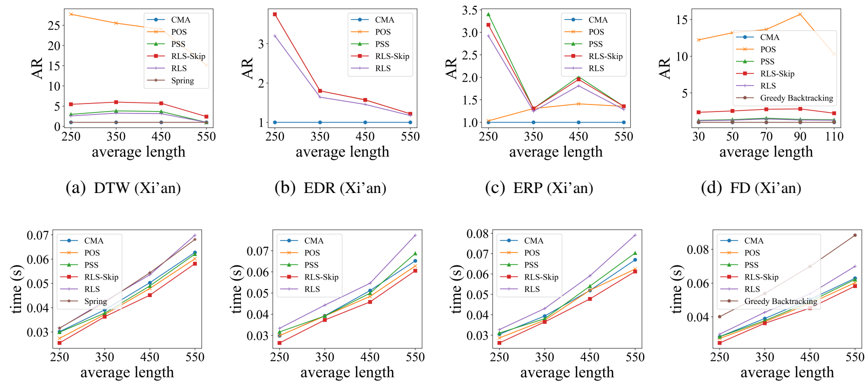
Metrics. We will compare our CMA algorithm with the existing algorithms regarding efficiency and effectiveness. For a given query trajectory, we evaluate the efficiency of an algorithm in terms of the time to find the most similar subtrajectory from all data trajectories. We use three evaluation metrics identical to those used in previous work to evaluate the solutions found by different algorithms in this experiment: (1) Approximate Ratio (AR). Given a distance function, AR represents the ratio of the distance between the query trajectory and the subtrajectory found by an approximate algorithm to the distance between the query trajectory and the optimal solution. (2) Mean Rank (MR). MR denotes the rank of the distance between the optimal subtrajectory found by the algorithm and the query trajectory among all subtrajectories of the original data trajectory. In particular, MR= 1 indicates that the algorithm finds the optimal solution. (3) Relative Rank (RR). RR is the percentage of all subtrajectories of the data trajectory that is better than the result returned by the algorithm. RR excludes the effect of trajectory length on MR.

Evaluation Platform. The methods are implemented in C++14. The experiments are conducted on a Linux server equipped with 48-cores of Intel(R) Xeon(R) 2.20GHz CPUs and with 128.00 GB RAM.

7.3 Experimental Results

Effectiveness compared with other algorithms We used different algorithms for each distance function in different datasets to find the subtrajectories of the data trajectories with the smallest distance from the query trajectory. The experimental results are shown in Table 4. The approximation algorithms have substantial uncertainty in terms of effectiveness. Although the subtrajectory found by these approximation algorithms when using ERP as the distance function is close to the optimal subtrajectory, the subtrajectory found by approximate algorithms when using DTW as the distance function is far from the optimal subtrajectory. POS and PSS tend to select the trajectories with the same length as the query trajectory due to the higher cost of deleting a point when ERP is used as the distance function. In contrast, the length of the optimal subtrajectory tends to vary when DTW is used as the distance function. In addition, the subtrajectories found by the RLS and RLS-Skip algorithms learned based on reinforcement learning are also far from the optimal subtrajectories. CMA can find the exact optimal solution in all cases.

Efficiency compared with other algorithms With the pruning algorithm, we can find the optimal subtrajectory from many data trajectories faster. Compared with ExactS, the efficiency of CMA has improved nearly 200 times on both Chengdu and Xi'an datasets and nearly 50 times on the Porto dataset according to the Table 5. The longer the length of the trajectory, the more the improvement of



(a) DTW (Xi'an) (b) EDR (Xi'an) (c) ERP (Xi'an) (d) FD (Xi'an)

(e) DTW (Xi'an) (f) EDR (Xi'an) (g) ERP (Xi'an) (h) FD (Xi'an)

Figure 7: Effectiveness and efficiency with varying data lengths

CMA over Exact. CMA can find the optimal subtrajectory relatively quickly regardless of the distance function. POS and RLS-Skip are the fastest, but they are approximate algorithms.

Effectiveness of the length of query trajectory In this experiment, we explore the effect of the length of the query trajectory on the efficiency and effectiveness of the algorithm by varying the length of the query trajectory. Figure 6 shows that the execution time of the algorithm increases with the length of the trajectory regardless of the dataset and distance function because the search algorithm takes less time to find the optimal subtrajectory for each query trajectory when the trajectory length is small. However, in the dataset of Porto, the execution time increases and then decreases with the length of the query trajectory, which may be attributed to the fact that there are fewer trajectories similar to the query trajectory in the dataset when the size of the query trajectory becomes longer. Thus, most trajectories are screened out in the filtering phase, resulting in a decrease in the final search time. RLS takes more time than other algorithms in almost all cases. All algorithms except CMA have poor performance when DTW is used as the distance function, regardless of the length of the query trajectory. It is because that DTW allows different points in query trajectory matches the same point in data trajectory. Furthermore, the effectiveness of the approximation algorithm is improved as the length of the query trajectory increases when EDR is used as the distance function. As the query trajectory length increases, the number of eligible data trajectories decreases, thus the approximation algorithm has a higher probability of finding the optimal solution. Both algorithms, RLS and RLS-Skip, also find much worse subtrajectories than MA, where RLS has a worse execution time than CMA in almost all cases.

Effectiveness of the length of data trajectories We test algorithm efficiency and effectiveness variation with the increasing length of data trajectories when using different distance functions on the Xi'an city dataset. In the experiment, we selected 2000 trajectories each with lengths in the intervals [200,300], [300,400], [400,500], and [500,600] from all trajectories in Xi'an city. Figure 7 shows that the time to find the optimal solution increases linearly with the length of the data trajectory for all algorithms. Moreover, Figure 7 also shows the relationship between the distance of subtrajectories and query trajectories found by different algorithms and the length of data trajectories. We find that the effectiveness of all approximate algorithms decreases with the increase of the length of data trajectories, which is because the number of subtrajectories increases when the length of data trajectories is large. Thus, the approximate algorithms are difficult to find out the most similar subtrajectory. Only CMA algorithm can still find the optimal subtrajectories among the data trajectories when the length of data trajectories increase. On the other hand, the time overhead increases linearly with the length of the data trajectory for all distance functions.

Performance of Spring and Greed Backtracking In this paper, we also explore the performance of Spring and Greedy Backtracking; the experimental results of Spring are shown in Figure 6(a) 6(f), while the results of Greedy Backtracking are shown in Figure 6(s) 6(x). The experimental results show that the AR of Spring and Greedy Backtracking is 1 in all cases, which means that both algorithms can find the optimal solution. However, in terms of efficiency, the execution time of Spring is similar to that of CMA, while Greedy Backtracking is less efficient.

7.4 Summary of Results

We verify that CMA can accurately find the nearest subtrajectory from the data trajectory to the query trajectory. Meanwhile, the execution time of CMA is about the same as the two approximation methods (i.e., PSS and POS), and is much smaller than ExactS. Therefore, the proposed algorithm can quickly and accurately find the subtrajectories of the closest data trajectory for each query trajectory.

8 CONCLUSION

This paper focuses on a similar subtrajectory search problem, i.e., finding the subtrajectory of the data trajectory with the minimum distance for the query trajectory. We convert the problem of finding the optimal subtrajectory to finding the optimal matching sequence. For a given query trajectory of length m and a data trajectory of length n, we propose the CMA algorithm to find the subtrajectory with the minimum distance to the query trajectory from the data trajectory in the time complexity of $O(mn)$. Finally, we conduct sufficient experiments on the datasets of Xi'an, Chengdu and Porto, and the experimental results show that our CMA algorithm can find efficiently the exact optimal subtrajectory for each query trajectory.

REFERENCES

[1] 2023. [Online] Technical Report. https://cspcheng.github.io/pdf/SubtrajectorySimilarity.pdf.
[2] Pankaj K Agarwal, Kyle Fox, Kamesh Munagala, Abhinandan Nath, Jiangwei Pan, and Erin Taylor. 2018. Subtrajectory clustering: Models and algorithms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 75–87.
[3] Helmut Alt and Michael Godau. 1995. Computing the Fréchet distance between two polygonal curves. *Int. J. Comput. Geom. Appl.* 5 (1995), 75–91.
[4] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. 2011. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications* 21, 03 (2011), 253–282.
[5] Lei Chen and Raymond T. Ng. 2004. On The Marriage of Lp-norms and Edit Distance. In *VLDB*. Morgan Kaufmann, 792–803.
[6] Lei Chen, M. Tamer Özsu, and Vincent Oria. 2005. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD Conference*. ACM, 491–502.
[7] Zaiben Chen, Heng Tao Shen, and Xiaofang Zhou. 2011. Discovering popular routes from trajectories. In *ICDE*. IEEE Computer Society, 900–911.
[8] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. *Acm Sigmod Record* 23, 2 (1994), 419–429.
[9] Joachim Gudmundsson, Martin P. Seybold, and John Pfeifer. 2021. On Practical Nearest Sub-Trajectory Queries under the Fréchet Distance. In *SIGSPATIAL/GIS*. ACM, 596–605.
[10] Bo Hui, Da Yan, Haiquan Chen, and Wei-Shinn Ku. 2021. TrajNet: A Trajectory-Based Deep Learning Model for Traffic Prediction. In *KDD*. ACM, 716–724.
[11] Bo Hui, Da Yan, Haiquan Chen, and Wei-Shinn Ku. 2021. Trajectory WaveNet: A Trajectory-Based Model for Traffic Forecasting. In *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE. https://doi.org/10.1109/icdm51629.2021.00131
[12] Satoshi Koide, Yukihiro Tadokoro, Takayoshi Yoshimura, Chuan Xiao, and Yoshiharu Ishikawa. 2018. Enhanced Indexing and Querying of Trajectories in Road Networks via String Algorithms. *ACM Trans. Spatial Algorithms Syst.* 4, 1 (2018), 3:1–3:41.
[13] Satoshi Koide, Chuan Xiao, and Yoshiharu Ishikawa. 2020. Fast Subtrajectory Similarity Search in Road Networks under Weighted Edit Distance Constraints. *Proc. VLDB Endow.* 13, 11 (2020), 2188–2201.
[14] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. 2007. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 593–604.
[15] Xiucheng Li, Kaiqi Zhao, Gao Cong, Christian S Jensen, and Wei Wei. 2018. Deep representation learning for trajectory similarity computation. In *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 617–628.
[16] Wuman Luo, Haoyu Tan, Lei Chen, and Lionel M. Ni. 2013. Finding time period-based most frequent path in big trajectory data. In *SIGMOD Conference*. ACM, 713–724.
[17] online. 2016. GAIA Open Dataset. https://outreach.didichuxing.com/appEn-vue/ChengDuOct2016?id=7.
[18] online. 2016. Porto Dataset. https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i/data.
[19] Sayan Ranu, Padmanabhan Deepak, Aditya D Telang, Prasad Deshpande, and Sriram Raghavan. 2015. Indexing and matching trajectories under inconsistent sampling rates. In *2015 IEEE 31st International conference on data engineering*. IEEE, 999–1010.
[20] Yasushi Sakurai, Christos Faloutsos, and Masashi Yamamuro. 2007. Stream Monitoring under the Time Warping Distance. In *ICDE*. IEEE Computer Society, 1046–1055.
[21] Panagiotis Tampakis, Christos Doulkeridis, Nikos Pelekis, and Yannis Theodoridis. 2020. Distributed subtrajectory join on massive datasets. *ACM Transactions on Spatial Algorithms and Systems (TSAS)* 6, 2 (2020), 1–29.
[22] Michail Vlachos, Dimitrios Gunopulos, and George Kollios. 2002. Discovering Similar Multidimensional Trajectories. In *ICDE*. IEEE Computer Society, 673–684.
[23] Jiachuan Wang, Peng Cheng, Libin Zheng, Chao Feng, Lei Chen, Xuemin Lin, and Zheng Wang. 2020. Demand-Aware Route Planning for Shared Mobility Services. *Proc. VLDB Endow.* 13, 7 (2020), 979–991.
[24] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Zizhe Xie, Qizhi Liu, and Xiaolin Qin. 2018. Torch: A Search Engine for Trajectory Data. In *SIGIR*. ACM, 535–544.
[25] Yilun Wang, Yu Zheng, and Yexiang Xue. 2014. Travel time estimation of a path using sparse trajectories. In *KDD*. ACM, 25–34.
[26] Zheng Wang, Cheng Long, Gao Cong, and Ce Ju. 2019. Effective and Efficient Sports Play Retrieval with Deep Representation Learning. In *KDD*. ACM, 499–509.
[27] Zheng Wang, Cheng Long, Gao Cong, and Yiding Liu. 2020. Efficient and Effective Similar Subtrajectory Search with Deep Reinforcement Learning. *Proc. VLDB Endow.* 13, 11 (2020), 2312–2325.
[28] Robert Waury, Christian S. Jensen, Satoshi Koide, Yoshiharu Ishikawa, and Chuan Xiao. 2019. Indexing Trajectories for Travel-Time Histogram Retrieval. In *EDBT*. OpenProceedings.org, 157–168.
[29] Min Xie. 2014. EDS: a segment-based distance measure for sub-trajectory similarity search. In *SIGMOD Conference*. ACM, 1609–1610.
[30] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. 1998. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *ICDE*. IEEE Computer Society, 201–208.
[31] Haitao Yuan and Guoliang Li. 2019. Distributed In-memory Trajectory Similarity Search and Join on Road Network. In *ICDE*. IEEE, 1262–1273.

APPENDIX

A PRUNING ALGORITHM

In the previous section, we proposed an algorithm with the time complexity of $O(mn)$ to find the subtrajectory with the minimum distance for the query trajectory in a data trajectory. The computation complexity for solving the SSS problem is thus optimized to $O(Nmn)$. Although the Algorithm 2 is efficient enough to complete a search within $10ms$, considering many data trajectories in the database (i.e., usually millions of data), the time to find the optimal subtrajectory from the database is about 15 minutes to half an hour for a given query trajectory. This is still not affordable for most of the applications. In practice, most of the data trajectories in the database are far distant from the query trajectory. We assume that only Q data trajectories are "similar" for each query trajectory when the number of data trajectories is N, then it is meaningless to search for the optimal subtrajectories from other data trajectories.

Filter with Key Points (FKP) For each point $\tau_q[i]$ in the query trajectory τ_q, we can convert it to a point in the data trajectory by substitution or deletion. Without considering other points, each point has a minimum cost of conversion. We use $minCost(\tau_q[i], \tau_d)$ to denote the lower bound of the cost of converting $\tau_q[i]$ to a point in τ_d. We give a formal definition of $minCost(\tau_q[i], \tau_d)$ as follows.

$$minCost(\tau_q[i], \tau_d) = \min\{del(\tau_q[i]), \min_{1\le j\le n} sub(\tau_q[i], \tau_d[j])\}$$

Next, we introduce the lower bound on the conversion cost by the Theorem A.1.

Theorem A.1 (Lower Bound of Cost). *We denote $minCost(\tau_q, \tau_d)$ as $\sum_{i=1}^{m} minCost(\tau_q[i], \tau_d)$ and $minCost(\tau_q, \tau_d)$ is the lower bound on the cost of converting τ_q to τ_d, i.e., $minCost(\tau_q, \tau_d) \le \min_{1\le j\le n} C_{m,j}$.*

PROOF. We use WED as an example to prove this theorem using mathematical induction.

(i) When $i = 1$, we have $minCost(\tau_q[1:i], \tau_d) \le \min_{1\le j\le n} sub(\tau_q[1], \tau_d[j]) = \min_{1\le j\le n} C_{m,j}$.

(ii) Suppose when $i = k-1$, we have $minCost(\tau_q[1:k-1], \tau_d) \le \min_{1\le j\le n} C_{k-1,j}$. We will discuss the case when $i = k$ in two scenarios. If $j = 1$, then $minCost(\tau_q[1:k], \tau_d) = minCost(\tau_q[1:k-1], \tau_d) + minCost(\tau_q[k], \tau_d)$. Considering that $minCost(\tau_q[1:k-1], \tau_d) < C_{k-1,1}$ and $minCost(\tau_q[1:k-1], \tau_d) < del(\tau_q[1:i-1])$, we have $minCost(\tau_q[1:k], \tau_d) \le \min\{C_{i-1,j} + del(\tau_q[i]), sub(\tau_q[i], \tau_d[1]) + del(\tau_q[1:i-1])\} = C_{k,1}$. For each $\forall j \ne 1$, we have

$$\begin{aligned}
& minCost(\tau_q[1:k], \tau_d) \\
= \;& minCost(\tau_q[1:k-1], \tau_d) + minCost(\tau_q[k], \tau_d) \\
\le \;& \min_{1\le t\le j} C_{k-1,t} + sub(\tau_q[i], \tau_d[j]) \\
\le \;& C_{k,j}
\end{aligned}$$

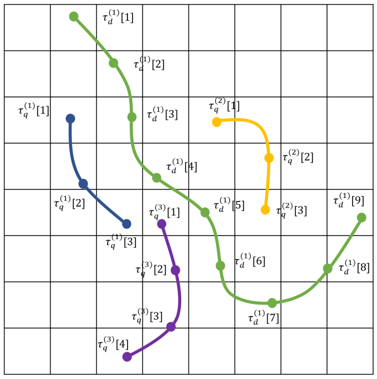


Figure 8: Pruning Example

Thus, we can obtain $minCost(\tau_q[1:k], \tau_d) \leq \min_{1 \leq j \leq n} C_{k,j}$.

Finally, by combining (i) and (ii), the theorem A.1 is proved. □

Nevertheless, we cannot directly compute $minCost(\tau_q, \tau_d)$ in practical applications. It is because the time complexity of computing $minCost(\tau_q, \tau_d)$ is $O(m \cdot n)$, which is the same as the time complexity of computing the optimal subtrajectory directly. Fortunately, we can approximate $minCost(\tau_q, \tau_d)$ due to the continuity of trajectory. In reality, the position of an object cannot change dramatically in a short time, thus the location of a point in a trajectory is close to the location of its neighboring points. Based on the continuity of the trajectory, we can select key points $Key(\tau_q)(= \{\tau_q[e_1], \tau_q[e_2], \ldots, \tau_q[e_K]\})$ from the original query trajectory. Based on these key points, we can compute the estimation of $minCost(\tau_q, \tau_d)$ and denote it by $minCost_e(\tau_q, \tau_d)$ as follows.

$$minCost_e(\tau_q, \tau_d) = r \sum_{\tau_q[e_i] \in Key(\tau_q)} minCost(\tau_q[e_i], \tau_d) \qquad (8)$$

Here, $r = \frac{|Key(\tau_d)|}{n}$ denoting the ratio of key points selected from the query trajectory. The selection of key points $Key(\tau_q)$ affects the accuracy of estimation; we uniformly select points from τ_q in this paper. The more key points in $Key(\tau_q)$, the more accurate the estimation of $minCost(\tau_q, \tau_d)$ will be. However, the time complexity of the computation also increases as $Key(\tau_q)$ increases. Therefore, we need to choose as few key points as possible while ensuring the estimation accuracy as much as possible.

Grid Based Pruning (GBP) The basic idea is to divide the whole map into multiple grids, and we use a hash table to record the grid where the key points of each trajectory are located. Then, for each point in the data trajectory, we only check all the key points of the query trajectory located within its grid and neighbor grids using the hash table. Finally, if a query trajectory has multiple key points around a given data trajectory, it is similar to the current data trajectory, and it is necessary to search the current data trajectory. Let us explain the details of the algorithm in the following part.

We divide the map into a square grid with ε as side lengths. For a point $\tau_d^{(k)}[j]$ in the data trajectory, we denote the grid it locates in as $g(\tau_d^{(k)}[j])$. Each grid is surrounded by 8 neighboring grids; we denote the set consisting of the grid $g(\tau_d^{(k)}[j])$ and its neighboring grids as $B(\tau_d^{(k)}[j])$. We consider $\tau_d^{(k)}[j]$ to be *close* to the points in the grid it is located in and its neighboring grids, i.e., a point $\tau_q^{(t)}[i]$ is close to $\tau_d^{(k)}[j]$ if and only if $g(\tau_q^{(t)}[i]) \in B(\tau_d^{(k)}[j])$. If there exists a point in a query trajectory that is close to $\tau_d^{(k)}[j]$, then we consider this query trajectory to be close to $\tau_d^{(k)}[j]$. Meanwhile, we denote the set formed by the points of all query trajectories close to $\tau_d^{(k)}[j]$ as $H(\tau_d^{(k)}[j])$.

$$H(\tau_d^{(k)}[j]) = \{\tau_q^{(t)}[i] \mid g(\tau_q^{(t)}[i]) \in B(\tau_d^{(k)}[j])\} \qquad (9)$$

Given a data trajectory $\tau_d^{(k)}$, we can count how many points in the query trajectory τ_q are close to that query trajectory and denote it as $close(\tau_q, \tau_d^{(k)})$, that is,

$$close(\tau_q, \tau_d^{(k)}) = \left| \left\{ \tau_q[i] \,\middle|\, \tau_q[i] \in \bigcup_{1 \leq j \leq n} H(\tau_d^{(k)}[j]) \right\} \right| \qquad (10)$$

The larger $close(\tau_q, \tau_d^{(k)})$ means that more points of the query trajectory are distributed around the data trajectory $\tau_d^{(k)}$, then there is likely a segment of sub-trajectories in $\tau_d^{(k)}$ that are similar to τ_q. We define a constant μ $(0 < \mu < 1)$ and we call the algorithm 2 to search for the optimal sub-trajectory in $\tau_d^{(k)}$ whenever $close(\tau_q, \tau_d^{(k)}) \geq \mu \cdot m$.

Complexity. By using the hash table, we can find the points in its neighboring grid for each $\tau_d^{(k)}[j]$ in a constant time. Thus the time complexity of computing $H(\tau_d^{(k)}[j])$ is $O(1)$, making the time complexity of computing for each point in the data trajectory $O(n)$.

Algorithm 3: Pruning Algorithm

Input: a query trajectory τ_q, data trajectories $\{\tau_d^{(1)}, \tau_d^{(2)}, \ldots, \tau_d^{(N)}\}$
Output: the optimal subtrajectory $\tau_d^{(k^*)}[i^* : j^*]$ for the query trajectory τ_q

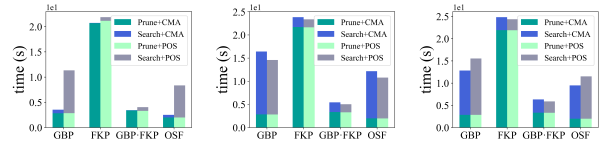
$\tau_d^{(*)}[i^* : j^*] \leftarrow \{\}$
forall $1 \leq k \leq N$ **do**
  For each point in the data trajectory $\tau_d^{(k)}$, we compute $H(\tau_d^{(k)}[j])$ according to the Equation 9
  According to the equation 10, we can compute $close(\tau_q, \tau_d^{(k)})$ for each query trajectory τ_q
  forall $1 \leq t \leq M$ **do**
    if $result = (-1, -1, -1)$ **then**
      $\tau_d^{(k)}[i^k : j^k] \leftarrow CMA(\tau_q, \tau_d^{(k)})$
      $\tau_d^{(*)}[i^* : j^*] \leftarrow \tau_d^{(k)}[i^k : j^k]$
    else
      if $close(\tau_q, \tau_d^{(k)}) > \mu \cdot m$ **then**
        calculate the estimation of the lower bound $minCost_e(\tau_q, \tau_d^{(k)})$ by Equation 8
        if $minCost_e(\tau_q, \tau_d^{(k)}) < \Theta(\tau_q, \tau_d^{(*)}[i^* : j^*])$ **then**
          $\tau_d^{(k)}[i^k : j^k] \leftarrow CMA(\tau_q, \tau_d^{(k)})$
          if $\Theta(\tau_q, \tau_d^{(k)}[i^k : j^k]) < \Theta(\tau_q, \tau_d^{(*)}[i^* : j^*])$ **then**
            $\tau_d^{(*)}[i^* : j^*] \leftarrow \tau_d^{(k)}[i^k : j^k]$
return $\tau_d^{(*)}[i^* : j^*]$

On the other hand, the computational complexity of Equation 10 depends mainly on the number of points in the query trajectory around the data trajectory, which is much smaller than n. Therefore, the pruning complexity is $O(Nn)$ in total if the average length of the data trajectories is n. Suppose the average length of the query trajectory is m. Since we select key points from the query trajectory at a certain ratio r, the number of key points is mr. Thus, the time complexity of computing the distance lower bound is $mrQn$, where Q denote the count of trajectories satisfying $close(\tau_q, \tau_d^{(k)}) > \mu \cdot m$. Finally, the time complexity of the algorithm 3 is $O(Nn + mrQn + Q'mn)$, where Q' denotes the count of trajectories satisfying line 12, and we have $N \gg Q \gg Q'$.

B EXPERIMENTAL RESULTS OF PRUNING ALGORITHM

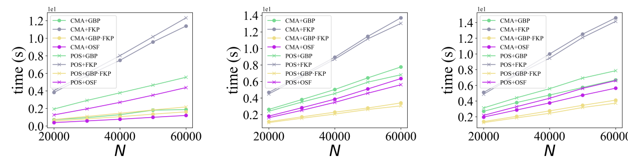
Analysis of pruning time and searching time. We explored the time of the pruning process and the search process with different pruning algorithms and search algorithms. We can find that the two modules, GBP and FKP, have different effects on the overall process of finding the optimal subtrajectories. For example, Figure 9 shows that using the GBP module will significantly speed up the pruning process because we only need a time complexity of $O(n)$ to determine whether this data trajectory τ_d is similar to the query trajectory. However, the drawback of GBP is that the fixed-parameter μ is not sufficient to sieve out all the dissimilar data trajectories well, which increases the number of invocations of the search algorithm. In particular, the time to complete the whole search process increases significantly when the complexity of the search algorithm is high.

On the other hand, Figure 9 shows that using FKP makes the pruning process significantly more time-consuming since no data structure is used to speed up the operation resulting in that we have to calculate the distance between the key point in the trajectory and the nearest point in the data trajectory for each query trajectory to estimate the lower bound of the distance between the optimal subtrajectory and the query trajectory. FKP will filter out all the data trajectories whose lower bound of distance from the query trajectory is greater than the distance between the current optimal subtrajectory and the query trajectory. The distance between the optimal subtrajectory and the query trajectory decreases as the search process continues, which means that more and more data trajectories will be sieved out, leaving only a few data trajectories that need to be searched. The results shown in Figure 9 indicate that the filtering process takes more time than the case of using only GBP when we use FKP, yet the search time is less than the case of using only GBP. Therefore, both the search and pruning processes take less time when we include both the GBP and the FKP module in the pruning process. In addition, the difference in pruning process overhead between using both FKP and GBP modules and using only GBP module is not significant, which indicates that using GBP module can significantly reduce the execution time of FKP module. Compared with OSF, GBP and KPF are able to sieve out more data trajectories that are not similar to the query trajectory, which also makes their search process much less time-consuming than OSF. In particular, the experimental results demonstrate that the search time using the CMA is significantly less than that using POS as the search algorithm when we use DTW as the distance function, which verifies the efficiency of the algorithm proposed in this paper.



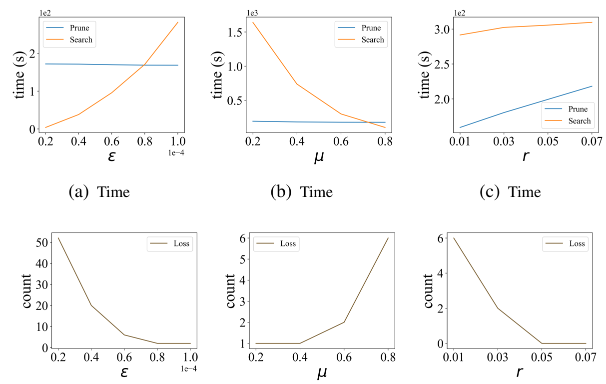
(a) DTW (b) EDR (c) ERP

Figure 9: Efficiency of Pruning and Searching



(a) DTW (b) EDR (c) ERP

Figure 10: Efficiency with varying data trajectories size N



(a) Time (b) Time (c) Time

(d) Loss (e) Loss (f) Loss

Figure 11: Effect of different parameters

Effect of the number of data trajectories on cost time. Similarly, we also investigate the relationship between the algorithm execution time and the number of data trajectories. As shown in Figure 10, the time of the whole query process shows a linear relationship with N. The pruning time does not increase significantly with the number of data trajectories when we use both the GBP and FKP modules, which shows the excellent scalability of our algorithm. The reason is that using the GBP module to filter the N data trajectories is fast. The increase in time is mainly since the number of data trajectories similar to the query trajectory increases as N becomes more extensive. In addition, the time of the whole query process increases obviously when using only the GBP module or the FKP module. Besides, the experimental results still show that the GBP·FKP proposed in this paper is superior to the effect of OSF. The time overhead of the pruning process while only using GBP or FKP is greater than the time overhead of the OSF algorithm.

Effect of r, ε and μ on pruning algorithm. The pruning condition setting can significantly impact the final pruning effect. Most of the trajectories can pass the pruning condition when the pruning condition is set loosely, which will lead to a sharp increase in the time to calculate the optimal subtrajectory. In contrast, when the strict pruning condition will lead to filtering out data trajectories similar to the query trajectory; thus the algorithm cannot find the optimal subtrajectory for some query trajectories. Therefore, two metrics (i.e., time and loss) are used to determine the impact of these three parameters r, ε, and μ on the pruning algorithm. A shorter run time of the algorithm means that the algorithm can filter out more trajectories that are not similar, while loss indicates the number of query trajectories for which the optimal subtrajectory cannot be found with the current parameter settings. We conducted experiments on the Xi'an dataset with different parameter settings based on ERP as a distance function using the MA algorithm, and the experimental results are shown in Figure 11. Figure 11(a) indicates that an increase in grid area leads to an increase in execution time through the algorithm. The reason is that a larger grid area causes more trajectories to pass through the filter and thus makes the algorithm execution less efficient. However, the increase in grid area will also reduce the probability of not finding the optimal subtrajectory for the query trajectory. Figure 11(b) shows that the execution efficiency of the algorithm decreases as μ increases. The larger μ means that more points in the query trajectory need to appear near the data trajectory before considering the data trajectory similar to the query trajectory. This results in more query trajectories not being able to find similar data trajectories as shown in Figure 11(e). r denotes the proportion of key points selected from the query trajectory, and a larger r indicates a more accurate estimate of the lower bound on the error. The experimental results shown in Figure 11(f) indicate that the loss decreases as r increases from 0.05 to 0.2, but the time required for the algorithm execution process increases.